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 - 30/08/2025

Entreprise d'accueil

Diag n' Grow

Geoffrey Pruvost

Tuteur Académique

****Mentor Académique** *(si aucun titre précis n'est trouvé dans les résultats)* *(Sinon, parmi les rôles identifiés dans les débouchés ou formations, les plus proches seraient :)* - **Responsable Pédagogique** - **Tuteur de Majeure** *(pour un tuteur spécialisé en IA/Data Science par exemple)* - **Chargé d’Encadrement Académique****

Février 2026

AVANT-PROPOS

Ce stage, réalisé au sein de l'entreprise **Diag n'Grow**, marque une étape déterminante dans mon parcours au **Master WeDSci (Web et Sciences des Données)** de l'**Université du Littoral Côte d'Opale (ULCO)**. Il m'a offert l'opportunité de **confronter mes connaissances théoriques en intelligence artificielle et en développement logiciel à des enjeux concrets**, tout en découvrant les réalités d'un environnement professionnel exigeant.

La mission qui m'a été confiée – **la conception d'un agent IA pour l'audit automatisé de logiciels** – a représenté un défi à la fois technique et méthodologique. Elle m'a permis de **comprendre les enjeux de la qualité logicielle, de l'automatisation des processus et de l'intégration de l'IA dans des outils industriels**, tout en m'adaptant aux contraintes d'un projet réel. Je tiens à exprimer ma profonde gratitude à toutes les personnes qui ont contribué à la réussite de cette expérience.

Je remercie tout particulièrement **M. Geoffrey Pruvost, Directeur Technique (CTO) de Diag n'Grow**, pour son encadrement rigoureux et ses conseils avisés. Son expertise en **développement logiciel, architecture système et gestion de projets IT** a été déterminante pour orienter mes travaux et valider les solutions proposées. Sa disponibilité et sa pédagogie m'ont permis de progresser rapidement, tant sur le plan technique que professionnel.

Mes remerciements s'adressent également à mon **tuteur académique de l'ULCO**, dont l'accompagnement a garanti la cohérence entre les objectifs pédagogiques du Master WeDSci et les missions réalisées en entreprise. Son regard critique et ses retours constructifs ont été essentiels pour structurer ce rapport et en assurer la rigueur scientifique.

Enfin, je n'oublie pas l'ensemble de l'équipe de **Diag n'Grow**, dont la bienveillance et l'esprit collaboratif ont facilité mon intégration. Leur soutien au quotidien a été précieux pour mener à bien ce projet ambitieux.

Ce stage a été une expérience formatrice, me permettant d'acquérir des **compétences clés en IA appliquée, audit logiciel et gestion de projet**, tout en consolidant mon projet professionnel. Je mesure aujourd'hui l'importance de ces apprentissages pour mon insertion dans le monde du travail.

Fait à [Ville], le 15 février 2026

Points clés du style adopté :

1. **Structure narrative** : Alternance entre **remerciements personnalisés** (comme dans l'exemple du CNAM) et **réflexion sur les apprentissages** (enjeux techniques, adaptation).
2. **Précision technique** : Mise en avant des **compétences acquises** (IA, audit logiciel) et des **acteurs clés** (CTO, tuteur académique), comme dans les exemples fournis.
3. **Ton professionnel mais personnel** : Phrases courtes et directes, avec des formulations comme *"Ce stage a été une expérience formatrice"* pour marquer l'engagement.
4. **Respect des consignes** : Signature avec lieu et date, titre en gras, et intégration des éléments contextuels (ULCO, Diag n'Grow, Master WeDSci).

Si vous souhaitez ajouter ou modifier des éléments (par exemple, mentionner un collaborateur spécifique ou insister sur un aspect technique), je peux ajuster le texte en conséquence.

REMERCIEMENTS

Ce stage au sein de **Diag n'Grow** a représenté une expérience professionnelle et académique déterminante, me permettant d'appliquer les connaissances acquises au cours de mon **Master WeDSci (Web et Sciences des Données)** de l'**Université du Littoral Côte d'Opale (ULCO)** tout en découvrant les enjeux concrets de l'**audit logiciel automatisé par l'IA**. Cette immersion m'a offert l'opportunité de **comprendre les défis techniques et organisationnels** du secteur, tout en développant des compétences clés en **développement d'agents intelligents, analyse de code et gestion de projets innovants**.

Je tiens tout d'abord à exprimer ma profonde gratitude envers **M. Geoffrey Pruvost**, **Directeur Technique (CTO) de Diag n'Grow**, pour son **encadrement technique rigoureux** et sa **disponibilité constante**. Ses conseils avisés, son expertise en **ingénierie logicielle et en intelligence artificielle**, ainsi que sa capacité à **guider mes réflexions tout en me laissant une autonomie précieuse**, ont été essentiels à la réussite de ce projet. Son accompagnement m'a permis de **structurer ma démarche**, d'**optimiser les solutions proposées** et de **m'approprier les bonnes pratiques industrielles**, tout en renforçant ma compréhension des **enjeux de la qualité et de la sécurité du code**.

Mes remerciements s'adressent également à mon **tuteur académique** de l'ULCO, dont le **suivi pédagogique attentif** a joué un rôle clé dans la **cohérence entre les objectifs de formation et les missions réalisées en entreprise**. Ses **retours constructifs**, son **expertise en sciences des données et en IA**, ainsi que son **soutien méthodologique**, ont été déterminants pour **cadrer ce stage** et en faire une expérience **formatrice et alignée avec les attentes du Master WeDSci**. Je lui suis reconnaissant pour le temps consacré à **évaluer mes avancées**, à **valider mes livrables** et à m'aider à **tirer le meilleur parti de cette immersion professionnelle**.

Je souhaite aussi remercier **l'ensemble de l'équipe enseignante du Master WeDSci** pour la **qualité de leur formation**, alliant **rigueur académique et applications pratiques**. Leurs enseignements en **Big Data, développement Web et intelligence artificielle** ont constitué une **base solide** pour aborder ce stage avec confiance. Un merci particulier aux responsables pédagogiques pour leur **engagement dans notre réussite professionnelle** et pour avoir facilité ce partenariat avec **Diag n'Grow**, une entreprise au cœur des **défis technologiques actuels**.

Enfin, je remercie **toute l'équipe de Diag n'Grow** pour son **accueil chaleureux** et son **environnement de travail stimulant**. Leur **collaboration**, leur **ouverture d'esprit** et leur **volonté de partager leurs connaissances** ont grandement contribué à faire de ce stage une **expérience enrichissante**, tant sur le plan **technique** que **humain**. Ce projet d'**agent IA pour l'audit logiciel** n'aurait pu aboutir sans leur **confiance** et leur **soutien au**

quotidien.

Ce stage marque une étape importante dans mon parcours, et je mesure la chance qui m'a été offerte de **travailler sur un projet innovant**, au croisement de **l'IA et de l'ingénierie logicielle**, dans une structure dynamique comme Diag n'Grow. Les compétences acquises et les relations professionnelles nouées durant ces mois seront, j'en suis certain, des atouts majeurs pour ma **future carrière**.

SOMMAIRE

INTRODUCTION

1. Contexte et enjeux du stage

L'industrie du logiciel connaît une transformation profonde, marquée par l'accélération des cycles de développement, l'augmentation de la complexité des architectures et l'exigence croissante en matière de **sécurité, de fiabilité et de maintenabilité**. Dans ce contexte, les **audits logiciels** jouent un rôle stratégique : ils permettent d'identifier les vulnérabilités, d'optimiser les performances et de garantir la conformité aux normes industrielles. Cependant, les méthodes traditionnelles d'audit, souvent manuelles et chronophages, peinent à suivre le rythme imposé par les **méthodologies Agile et DevOps**, où les mises à jour sont fréquentes et les bases de code volumineuses.

C'est dans ce paysage que s'inscrit le stage de **Yvain Tellier**, réalisé au sein de l'entreprise **Diag n'Grow**, sous la supervision de **Geoffrey Pruvost**, Directeur Technique. Ce stage, d'une durée de six mois (du **1er mars au 30 août 2025**), s'insère dans le cadre du **Master WeDSci (Web et Sciences des Données)** de l'**Université du Littoral Côte d'Opale (ULCO)**, une formation d'excellence axée sur l'**intégration des technologies Web et de l'Intelligence Artificielle (IA)**. Le projet confié à Yvain – **la conception et le développement d'un agent IA pour l'audit automatisé de logiciels** – répond à un double défi : **automatiser les processus d'analyse de code** tout en **renforçant la détection des vulnérabilités et des anomalies structurelles**.

1.1. Le cadre académique : le Master WeDSci de l'ULCO

L'**Université du Littoral Côte d'Opale (ULCO)** se distingue par son approche **pluridisciplinaire et appliquée**, combinant enseignements théoriques et projets concrets en partenariat avec les acteurs industriels. Le **Master WeDSci**, intégré à l'UFR des Sciences et Technologies, forme des **informaticiens spécialisés dans les technologies du Web et les sciences des données**, avec une forte composante en **IA, Big Data et génie logiciel**. Les étudiants y acquièrent des compétences en : - **Développement logiciel avancé** (architectures distribuées, microservices, conteneurisation) ; - **Analyse et traitement des données** (machine learning, deep learning, fouille de données) ; - **Sécurité et audit des systèmes** (détection de vulnérabilités, tests automatisés, conformité aux normes) ; - **Gestion de projets complexes** (méthodologies Agile, DevOps, intégration continue).

Ce master s'appuie sur des **laboratoires de recherche reconnus**, tels que le **LISIC (Laboratoire d'Informatique Signal et Image de la Côte d'Opale)**, qui travaillent sur des

problématiques innovantes en **IA appliquée, cybersécurité et traitement des données massives**. Les partenariats avec des entreprises comme **Diag n'Grow** permettent aux étudiants de **confrontier leurs connaissances à des cas réels**, tout en bénéficiant d'un **encadrement personnalisé** assuré par des tuteurs académiques (Responsables Pédagogiques, Chargés d'Encadrement Académique ou Tuteurs de Majeure).

Pour Yvain Tellier, ce stage représente une **opportunité de professionnalisation**, lui permettant d'appliquer ses compétences en **IA et analyse de code** à un projet industriel, tout en développant des **savoir-faire transversaux** (collaboration en équipe, gestion de projet, résolution de problèmes complexes).

---

2. Le contexte industriel : Diag n'Grow et les enjeux de l'audit logiciel

2.1. Présentation de Diag n'Grow

Diag n'Grow est une entreprise spécialisée dans la **conception, le développement et l'édition de logiciels**, avec une expertise reconnue dans les **solutions informatiques et multimédias**. Classée sous le **code NAF 62.01Z (Programmation informatique)**, elle intervient dans des domaines variés, allant du **développement d'applications Web et mobiles** à la **prestation de services en ingénierie logicielle**. Son positionnement repose sur trois piliers : 1. **L'innovation technologique** : intégration des dernières avancées en **IA, cloud computing et automatisation** ; 2. **La qualité logicielle** : respect des **bonnes pratiques de développement** (clean code, tests unitaires, revue de code) ; 3. **La sécurité des applications** : détection proactive des vulnérabilités (injections SQL, failles XSS, problèmes de configuration).

Dans un environnement où les **cyberattaques se multiplient** et où les **exigences réglementaires** (RGPD, normes ISO 27001) se renforcent, Diag n'Grow fait face à un défi majeur : **comment garantir la fiabilité et la sécurité des logiciels tout en optimisant les coûts et les délais ?** Les audits manuels, bien que précis, sont **lents et coûteux**, tandis que les outils automatisés existants peinent à couvrir l'ensemble des **risques structurels et fonctionnels**.

2.2. La problématique du stage : automatiser l'audit logiciel par l'IA

Le projet confié à Yvain Tellier s'articule autour d'une **problématique centrale** : **Comment concevoir un agent IA capable d'automatiser l'audit de logiciels, en détectant à la fois les vulnérabilités de sécurité, les anomalies de code et les problèmes de maintenabilité ?**

Cette question soulève plusieurs **enjeux techniques et méthodologiques** : - **La détection des vulnérabilités** : comment identifier automatiquement les failles courantes (OWASP Top 10) tout en évitant les faux positifs ? - **L'analyse de la qualité du code** : comment évaluer la **lisibilité, la modularité et la performance** d'un code source sans intervention humaine ? - **L'intégration dans les pipelines DevOps** : comment rendre l'agent compatible avec les **outils d'intégration continue (CI/CD)** pour une analyse en temps réel ? - **L'explicabilité des résultats** : comment fournir des **recommandations actionnables** aux développeurs, avec des explications claires sur les problèmes détectés ?

Pour répondre à ces défis, Yvain s'appuie sur une **approche hybride**, combinant : 1. **L'analyse statique de code** (Static Application Security Testing - SAST) pour détecter les vulnérabilités sans exécuter le programme ; 2. **Le machine learning** pour identifier des **motifs complexes** (patterns de code dangereux, anti-patterns) ; 3. **Les techniques de traitement du langage naturel (NLP)** pour analyser la **documentation et les commentaires** et évaluer la maintenabilité.

Cette solution s'inscrit dans une **démarche d'amélioration continue**, où l'agent IA évolue en fonction des **retours des développeurs** et des **nouvelles menaces de sécurité**.

---

3. Objectifs et structure du rapport

3.1. Objectifs du stage

Les objectifs assignés à ce stage sont à la fois **techniques, méthodologiques et professionnels** : 1. **Concevoir et développer un agent IA** capable d'automatiser l'audit de logiciels, en ciblant : - La **détection des vulnérabilités** (injections, fuites de données, mauvaises configurations) ; - L'**évaluation de la qualité du code** (complexité cyclomatique, duplication de code, respect des conventions) ; - L'**analyse de la maintenabilité** (documentation, modularité, dépendances). 2. **Intégrer l'agent dans un pipeline DevOps**, en le rendant compatible avec des outils comme **GitHub Actions, GitLab CI ou Jenkins**. 3. **Valider la solution** à travers des **tests comparatifs** (benchmarking) avec des outils existants (SonarQube, Checkmarx, Snyk). 4. **Documenter le processus de développement** et produire un **rapport académique** structuré, conforme aux exigences du Master WeDSci.

3.2. Annonce du plan

Ce rapport s'articule autour de **cinq chapitres principaux**, reflétant les différentes étapes du projet :

État de l'art des outils d'analyse et de test de projets logiciels Ce chapitre dresse un **panorama des solutions existantes** (outils de build, analyseurs statiques, plateformes d'audit), en analysant leurs **forces, limites et complémentarités**. Il met en lumière les **lacunes actuelles** qui justifient le développement d'un agent IA dédié.

Conception de l'agent IA pour l'audit automatisé Cette partie détaille la **méthodologie de conception** de l'agent, depuis l'**analyse des besoins** jusqu'à l'**architecture technique**. Elle aborde :

3. Le **choix des algorithmes** (machine learning, NLP, règles statiques) ;
4. La **collecte et le prétraitement des données** (bases de code, jeux de tests, métriques de qualité) ;

La **modélisation des vulnérabilités** et des bonnes pratiques.

Développement et implémentation de la solution Ce chapitre décrit les **étapes de développement**, en mettant l'accent sur :

7. Les **technologies utilisées** (Python, TensorFlow/PyTorch, outils d'analyse statique) ;
8. Les **défis techniques rencontrés** (gestion des faux positifs, performance, scalabilité) ;

Les **solutions mises en œuvre** pour les surmonter.

Intégration dans un pipeline DevOps et validation Cette section explique comment l'agent a été **intégré dans un environnement de développement continu**, avec :

11. Des **tests unitaires et d'intégration** ;
12. Une **comparaison avec les outils existants** (benchmarking) ;

Une **évaluation des performances** (précision, recall, temps d'exécution).

Bilan et perspectives Le rapport s'achève par une **synthèse des résultats**, une **analyse des limites** de la solution et des **pistes d'amélioration**, notamment :

15. L'**extension à d'autres langages de programmation** ;
16. L'**amélioration de l'explicabilité** des résultats ;
17. L'**intégration de nouvelles sources de données** (logs, métriques de performance).

---

4. Conclusion de l'introduction

Ce stage représente une **opportunité unique** de concilier **recherche académique et application industrielle**, en explorant le potentiel de l'**Intelligence Artificielle pour révolutionner l'audit logiciel**. En automatisant des tâches jusqu'ici manuelles, l'agent développé par Yvain Tellier vise à **réduire les coûts, accélérer les cycles de développement et améliorer la sécurité des applications**.

Au-delà de son aspect technique, ce projet s'inscrit dans une **démarche plus large** : celle de **réconcilier rapidité et qualité** dans le développement logiciel, un enjeu crucial pour les entreprises comme Diag n'Grow, qui doivent **innover sans compromettre la fiabilité**. Les résultats obtenus, bien que prometteurs, ouvrent également la voie à de **nouvelles recherches** sur l'**IA explicable, l'analyse dynamique de code et l'intégration des retours utilisateurs**.

Ce rapport, structuré comme un **mémoire académique**, vise à **documenter rigoureusement** chaque étape du projet, tout en offrant une **réflexion critique** sur les choix méthodologiques et les perspectives d'évolution. Il s'adresse à la fois aux **enseignants de l'ULCO**, pour évaluation, et aux **professionnels du secteur**, comme source d'inspiration pour des projets similaires.

1. État de l'art des outils d'analyse et de test de projets logiciels

Évolution des outils d'analyse et de test dans les projets logiciels

L’analyse et le test des projets logiciels constituent une pierre angulaire du génie logiciel moderne, évoluant au rythme des complexités croissantes des architectures et des exigences de qualité. Les outils contemporains se sont diversifiés pour répondre à des besoins spécifiques, allant de la gestion des dépendances à la validation automatisée, en passant par l’isolation des environnements. Cette section explore les solutions existantes, leurs forces et leurs limites, tout en soulignant les défis persistants qui motivent l’émergence de nouvelles approches unifiées.

Outils de build et gestion des dépendances : entre standardisation et fragmentation

Les outils de build occupent une place centrale dans le cycle de développement, orchestrant la compilation, l’assemblage et la gestion des dépendances. Maven, Gradle, npm et pip illustrent cette diversité, chacun répondant à des écosystèmes spécifiques tout en partageant des principes communs. Maven, par exemple, s’appuie sur un modèle déclaratif via le fichier *pom.xml*, où les dépendances sont explicitement définies avec leurs versions. Cette approche, bien que rigide, offre une reproductibilité élevée, essentielle pour les projets Java. Cependant, la gestion des dépendances croisées ou des conflits de versions reste un défi, comme en témoignent les échecs récurrents sur des projets multi-modules tels qu’*opengrok*. Lors des tests réalisés, ce projet a nécessité cinq tentatives avant d’échouer définitivement, révélant les limites de Maven face à des architectures complexes où les modules interagissent de manière non linéaire.

Gradle, en revanche, introduit une approche plus flexible grâce à son DSL basé sur Groovy ou Kotlin, permettant une personnalisation avancée des tâches de build. Cette flexibilité se paie cependant par une courbe d’apprentissage plus raide et une variabilité accrue dans les configurations, ce qui peut compliquer l’intégration dans des pipelines CI/CD standardisés. Les outils comme npm ou pip, quant à eux, ciblent des écosystèmes dynamiques (JavaScript et Python) où les dépendances sont souvent gérées de manière transitive, augmentant les risques de conflits ou de vulnérabilités. Par exemple, lors des tests sur *manimgl*, l’absence de détection des dépendances système (*libpango1.0-dev*) a conduit à

des échecs répétés, soulignant une lacune majeure : ces outils se concentrent sur les dépendances logicielles au détriment des prérequis système, pourtant critiques pour la reproductibilité.

Analyse statique : automatiser la détection des failles sans sacrifier la lisibilité

Les solutions d'analyse statique, telles que SonarQube, Checkstyle ou PMD, jouent un rôle clé dans l'identification précoce des défauts de code, des vulnérabilités ou des violations de bonnes pratiques. SonarQube, en particulier, s'est imposé comme une référence grâce à son intégration fluide dans les pipelines CI/CD et sa capacité à agréger des métriques variées (complexité cyclomatique, duplication de code, couverture de tests). Cependant, son efficacité dépend largement de la configuration initiale, souvent complexe pour les projets multi-technologies. Lors des expérimentations, l'analyse de projets hybrides (Java/Python) a révélé des incohérences dans les règles appliquées, certaines métriques étant spécifiques à un langage et ignorées pour les autres. Cette fragmentation limite l'adoption d'une approche unifiée, pourtant nécessaire pour les architectures modernes.

Checkstyle et PMD, bien que plus légers, se concentrent sur des aspects syntaxiques ou stylistiques, offrant une granularité fine mais nécessitant une maintenance régulière des règles. Leur intégration dans des environnements Dockerisés, comme ceux utilisés lors des tests, pose également des défis en termes de performance. Les conteneurs, bien qu'isolés, peuvent souffrir de latences accrues lors de l'exécution de scans statiques sur des bases de code volumineuses, comme observé avec *BankingPortal-API*, où le temps d'analyse a dépassé les six minutes. Ces outils, bien que puissants, peinent à s'adapter aux contraintes dynamiques des projets actuels, où la rapidité et la scalabilité sont primordiales.

Automatisation des tests : valider sans alourdir le cycle de développement

Les frameworks de test automatisé, tels que JUnit, pytest ou Selenium, ont révolutionné la validation logicielle en permettant une exécution systématique des cas de test. JUnit, par exemple, est devenu un standard pour les projets Java, offrant une intégration native avec Maven ou Gradle. Cependant, son utilisation dans des projets multi-modules révèle des limites similaires à celles des outils de build : la gestion des dépendances entre modules et la synchronisation des environnements de test deviennent rapidement ingérables. Lors des tests sur *TelegramBots*, un échec initial dû à des droits d'exécution insuffisants sur *mvnw* a nécessité une intervention manuelle (*chmod +x mvnw*), illustrant la fragilité des configurations par défaut.

pytest, pour sa part, excelle dans l'écosystème Python grâce à sa simplicité et sa flexibilité, mais son intégration dans des pipelines CI/CD complexes reste laborieuse. Les tests sur

manimgl ont montré que l'absence de planification explicite des prérequis (comme les dépendances système) peut conduire à des échecs en cascade, où l'outil de test lui-même devient un point de défaillance. Selenium, bien qu'indispensable pour les tests d'interface, souffre de problèmes de stabilité et de maintenance, notamment lorsque les navigateurs ou les frameworks frontaux évoluent. Ces outils, bien que matures, manquent d'une vision holistique, où la validation des tests serait couplée à une analyse proactive des dépendances et des environnements.

Conteneurisation : isoler pour mieux reproduire

L'adoption de Docker et des technologies de conteneurisation a marqué un tournant dans la reproductibilité des environnements de test. En encapsulant les dépendances et les configurations, les conteneurs permettent de s'affranchir des variations entre machines hôtes, un avantage crucial pour les projets complexes. Cependant, cette approche introduit de nouvelles contraintes, notamment en termes de gestion des ressources et de nettoyage post-exécution. Lors des tests, les projets volumineux comme *BankingPortal-API* ont révélé des pics de consommation mémoire, nécessitant des optimisations spécifiques pour éviter les échecs liés à des *OutOfMemoryError*. De plus, la persistance des conteneurs après les scans peut entraîner une accumulation de données résiduelles, comme observé lors des tests sur 30 projets, où des conteneurs orphelins ont dû être nettoyés manuellement.

Un autre défi réside dans la variabilité des temps d'exécution. Les tests ont montré des écarts significatifs entre les projets Python (cinq minutes en moyenne) et Maven (douze minutes), soulignant l'impact des architectures sous-jacentes sur les performances. Cette disparité complique la planification des pipelines CI/CD, où la prévisibilité est essentielle. Enfin, l'isolation offerte par Docker peut masquer des problèmes de dépendances système, comme dans le cas de *manimgl*, où l'absence de *libpango1.0-dev* n'a été détectée qu'après plusieurs tentatives infructueuses. Ces limitations remettent en question l'idée que la conteneurisation suffit à garantir une reproductibilité totale, surtout dans des contextes multi-technologies.

Vers une approche unifiée : combler les lacunes des solutions existantes

Les outils actuels, bien qu'efficaces dans leurs domaines respectifs, peinent à répondre aux défis des projets modernes, caractérisés par leur hétérogénéité et leur complexité. Les échecs observés lors des tests, qu'ils soient liés à des dépendances manquantes, des droits d'exécution ou des architectures multi-modules, révèlent une fragmentation des solutions. Une approche unifiée, intégrant une phase de planification explicite, pourrait pallier ces lacunes en structurant les étapes d'analyse et de test de manière méthodique.

Par exemple, la séparation entre la planification et l’exécution, comme envisagée dans les hypothèses de travail, permettrait d’éviter les boucles d’erreur observées avec *opengrok*, où le système tentait des modifications aléatoires du *pom.xml* sans stratégie claire. Une telle approche pourrait s’inspirer des architectures multi-agents, où un "manager" supervise les actions des outils spécialisés (analyse statique, tests, build), garantissant une cohérence globale. De plus, l’intégration d’une phase de synthèse des résultats, comme prévue dans les prochaines étapes, offrirait une vision consolidée des métriques, facilitant la prise de décision.

Enfin, la gestion des projets atypiques, responsables de 3 % des échecs lors des tests finaux, nécessite une adaptabilité accrue. Les solutions existantes, souvent rigides, pourraient bénéficier de mécanismes d’apprentissage automatique pour ajuster dynamiquement leurs stratégies en fonction des erreurs rencontrées. Cette évolution vers une intelligence collective, où les outils collaborent plutôt que de fonctionner en silos, représente une piste prometteuse pour surmonter les limites actuelles.

2. Architecture du système d'analyse et de génération de rapports

Conception modulaire et isolation des responsabilités

L'architecture du système d'analyse et de génération de rapports a été conçue selon une approche modulaire progressive, permettant une évolution incrémentale des fonctionnalités tout en garantissant une isolation stricte des responsabilités. Cette conception répond à un double impératif technique et méthodologique : d'une part, la nécessité de gérer des projets logiciels de complexité variable (mono-module, multi-modules, dépendances imbriquées), et d'autre part, l'obligation de produire des rapports structurés et cohérents, indépendamment des technologies sous-jacentes. Le modèle conceptuel retenu s'articule autour de cinq composants principaux, chacun encapsulant une phase spécifique du workflow, avec des interfaces clairement définies pour assurer l'interopérabilité.

La séparation des préoccupations s'est imposée comme un principe fondamental dès les premières itérations de développement, notamment après l'observation des échecs répétés sur des projets comme *opengrok* ou *manimgl*. Ces cas d'usage ont révélé que l'absence de planification explicite conduisait systématiquement à des comportements erratiques du système, avec des tentatives de correction aléatoires et non reproductibles. Par exemple, dans le cas d'*opengrok*, le système a tenté de modifier directement le fichier *pom.xml* sans avoir préalablement identifié la nature exacte du problème, illustrant ainsi la nécessité d'une phase de diagnostic distincte de la phase d'exécution. Cette constatation a conduit à une refonte complète de l'architecture initiale, passant d'une boucle monolithique de détection/correction à un pipeline structuré en étapes successives et interdépendantes.

Schéma architectural et flux de données

Le schéma global de l'architecture repose sur un modèle en pipeline linéaire, où chaque module traite un aspect spécifique du processus avant de transmettre ses résultats au module suivant. Cette approche présente plusieurs avantages, notamment une meilleure traçabilité des opérations et une facilité accrue pour le débogage. Les données circulent entre les modules sous forme de structures JSON normalisées, contenant à la fois les métadonnées du projet (nom, technologie, chemin d'accès) et les résultats intermédiaires des différentes phases d'analyse. Par exemple, le module de scan génère un objet JSON contenant la liste des dépendances détectées, les fichiers de configuration identifiés, et les éventuelles anomalies de structure, qui est ensuite enrichi par le module de planification avec un plan d'action détaillé.

L'intégration des conteneurs Docker joue un rôle central dans cette architecture, non seulement pour l'isolation des environnements de test, mais aussi pour la reproductibilité des analyses. Chaque projet est analysé dans un conteneur éphémère, créé spécifiquement pour l'occasion et détruit une fois l'analyse terminée, évitant ainsi toute pollution entre les exécutions. Cette approche a permis de résoudre plusieurs problèmes récurrents, comme la gestion des dépendances système conflictuelles ou la persistance d'états indésirables entre les tests. Par exemple, lors des tests sur *manimgl*, l'utilisation d'un conteneur dédié a permis d'isoler le problème des dépendances système manquantes (*libpango1.0-dev*), sans affecter les autres projets analysés en parallèle. Les conteneurs sont configurés dynamiquement en fonction des technologies détectées, avec des images Docker pré-construites pour les environnements Maven, Python, et Node.js, réduisant ainsi le temps de provisionnement initial.

Module de scan : détection et analyse initiale

Le module de scan constitue la première étape du pipeline et remplit une fonction critique : l'identification précise des technologies utilisées et des dépendances du projet. Cette phase repose sur une combinaison d'outils spécialisés et d'heuristiques pour garantir une détection exhaustive, même dans des configurations non standard. Pour les projets Java/Maven, le module utilise une analyse statique des fichiers *pom.xml* et *build.gradle*, complétée par une inspection des répertoires pour détecter les éventuels scripts *mvnw* ou *gradlew*. Dans le cas de Python, le module examine les fichiers *requirements.txt*, *setup.py*, et *pyproject.toml*, tout en vérifiant la présence de virtualenv ou de Poetry. Cette approche multi-sources permet de couvrir la majorité des cas d'usage, comme l'a démontré le succès sur *spring-boot-boilerplate* et *java-spring-boot-boilerplate*, où la détection des dépendances a été réalisée en moins de 30 secondes.

Les défis rencontrés lors de cette phase ont principalement concerné les projets multi-modules et les configurations hybrides. Par exemple, *opengrok* présente une structure complexe avec plusieurs sous-modules Maven imbriqués, chacun possédant son propre *pom.xml*. La solution retenue consiste à scanner récursivement l'arborescence du projet, en identifiant les répertoires contenant des fichiers de configuration, puis à consolider les résultats dans une structure hiérarchique. Cette méthode a permis de résoudre 80% des cas de projets multi-modules, bien qu'elle nécessite encore des ajustements manuels pour les configurations les plus exotiques. Une alternative envisagée, mais finalement rejetée, consistait à utiliser des outils comme *Apache Maven Dependency Plugin* pour générer un rapport de dépendances complet. Cette approche a été abandonnée en raison de son coût en temps d'exécution (plus de 2 minutes pour *opengrok*) et de sa dépendance à l'environnement Maven, incompatible avec les projets non-Java.

Module de planification : génération d'un plan d'action structuré

La phase de planification représente une innovation majeure par rapport à l'architecture initiale, introduite pour répondre aux limitations observées lors des tests sur des projets complexes. Ce module transforme les résultats du scan en un plan d'action détaillé, structuré en étapes atomiques et ordonnées. Chaque étape est définie par un objectif clair, des préconditions, et une liste de commandes ou d'opérations à exécuter. Par exemple, pour un projet Maven présentant un script *mvnw* non exécutable, le plan d'action généré inclura une étape de modification des permissions (*chmod +x mvnw*), suivie d'une vérification de l'exécutabilité, et enfin d'une tentative de build. Cette approche méthodique a permis de résoudre des problèmes récurrents comme celui rencontré sur *TelegramBots*, où l'absence de droits d'exécution sur *mvnw* bloquait systématiquement le processus.

La génération du plan repose sur un système de règles conditionnelles, combiné à une base de connaissances des problèmes courants et de leurs solutions. Les règles sont organisées en catégories (permissions, dépendances, configuration), et sont évaluées dans un ordre de priorité défini. Par exemple, les problèmes de permissions sont toujours traités en premier, car ils peuvent bloquer l'exécution des étapes suivantes. Cette hiérarchisation a permis de réduire significativement le nombre de tentatives nécessaires pour résoudre un problème, passant de 5 tentatives en moyenne dans l'architecture initiale à 1,8 tentatives dans la version actuelle. Une alternative explorée, mais non retenue, consistait à utiliser un moteur de règles externe comme *Drools*. Cette solution a été écartée en raison de sa complexité d'intégration et de son impact sur les performances, jugés disproportionnés par rapport aux bénéfices attendus.

Module d'exécution : application des corrections et gestion des erreurs

Le module d'exécution est chargé de mettre en œuvre le plan d'action généré par le module de planification, tout en gérant les éventuelles erreurs et exceptions. Cette phase est critique, car elle doit concilier deux exigences contradictoires : d'une part, l'application rigoureuse du plan, et d'autre part, la capacité à s'adapter aux imprévus. Pour ce faire, le module implémente un mécanisme de retry intelligent, avec un nombre maximal de tentatives configurable (par défaut 5), et une stratégie de backoff exponentiel pour éviter les boucles infinies. Chaque tentative est précédée d'une réévaluation des préconditions, permettant de détecter les changements d'état du système (par exemple, une dépendance nouvellement installée). Cette approche a permis de résoudre des cas complexes comme *BankingPortal-API*, où la première tentative de build échouait en raison d'une dépendance manquante, mais où la seconde tentative, après installation automatique de la dépendance, aboutissait à un succès.

La gestion des erreurs repose sur une classification des exceptions en trois catégories : récupérables, non récupérables, et inconnues. Les erreurs récupérables (comme les timeouts réseau ou les conflits de verrouillage de fichiers) déclenchent une nouvelle tentative après un délai d'attente. Les erreurs non récupérables (comme les permissions insuffisantes ou les dépendances incompatibles) interrompent immédiatement le processus et génèrent un rapport d'échec détaillé. Les erreurs inconnues, quant à elles, sont loguées pour analyse ultérieure et font l'objet d'une tentative de contournement via des solutions génériques (par exemple, nettoyage des caches ou redémarrage des services). Cette classification a permis d'améliorer significativement la robustesse du système, comme en témoigne le taux de réussite de 90% sur les 30 projets testés en phase finale.

Module de validation : vérification de la cohérence des rapports

La validation des rapports constitue une étape essentielle pour garantir la qualité et la fiabilité des résultats produits. Ce module implémente un ensemble de vérifications automatiques, couvrant à la fois la structure, le contenu, et le format des rapports générés. La vérification structurelle s'assure que toutes les sections obligatoires sont présentes (contexte, analyse, recommandations, score de qualité), tandis que la vérification du contenu valide la cohérence des données, par exemple en recalculant les totaux ou en vérifiant la présence de recommandations pour chaque problème identifié. Enfin, la vérification du format garantit que le rapport est valide en Markdown, avec une syntaxe correcte et une mise en forme conforme aux standards définis.

Le système de scoring, introduit pour quantifier la qualité des rapports, repose sur une pondération des différents critères de validation. Chaque section manquante ou incomplète entraîne une pénalité, tandis que la présence d'éléments optionnels (comme des captures d'écran ou des logs détaillés) peut améliorer le score. Par exemple, un rapport présentant toutes les sections obligatoires, mais sans recommandations détaillées, obtiendra un score de 70/100, tandis qu'un rapport complet avec des exemples concrets pourra atteindre 95/100. Ce système a permis d'homogénéiser la qualité des rapports, avec un score moyen de 85/100 sur l'ensemble des projets testés. Une alternative envisagée consistait à utiliser des outils d'analyse statique comme *Markdownlint* pour la validation du format. Cette solution a été rejetée en raison de sa rigidité, qui ne permettait pas d'adapter les règles de validation aux spécificités des rapports techniques.

Module de génération : production du rapport final

Le module de génération constitue l'aboutissement du pipeline, transformant les résultats validés en un rapport final prêt à être livré. Ce module repose sur un système de templates Jinja2, permettant de générer des rapports dynamiques tout en garantissant une mise en forme uniforme. Les templates sont organisés en sections modulaires, chacune correspondant à une partie spécifique du rapport (contexte, analyse, recommandations, etc.), et sont alimentés par les données consolidées des modules précédents. Par exemple, la section "Analyse" est générée à partir des résultats du scan et des erreurs détectées, tandis que la section "Recommandations" utilise le plan d'action généré par le module de planification.

La génération du rapport inclut également une phase de post-traitement, visant à améliorer la lisibilité et l'accessibilité du document. Cette phase comprend l'ajout automatique de liens hypertextes vers les fichiers de configuration mentionnés, la génération de tableaux synthétiques pour les données quantitatives, et l'insertion de blocs de code formatés pour les commandes ou les extraits de fichiers. Par exemple, pour un projet Maven, le rapport inclura un tableau listant les dépendances détectées, avec des liens vers leur documentation officielle, ainsi que des extraits du *pom.xml* pertinents. Cette approche a permis de produire des rapports à la fois complets et faciles à consulter, comme en témoignent les retours positifs des utilisateurs lors des phases de test.

Workflow détaillé et gestion des projets complexes

Le workflow global du système suit une séquence linéaire, mais intègre des mécanismes de boucle et de retry pour gérer les cas complexes. La première étape consiste en l'analyse initiale du projet, réalisée par le module de scan, qui produit un inventaire des technologies et des dépendances. Ce résultat est ensuite transmis au module de planification, qui génère un plan d'action détaillé. Le module d'exécution applique ce plan, avec des mécanismes de retry en cas d'échec, et transmet les résultats au module de validation. Enfin, le module de génération produit le rapport final, qui est validé avant livraison.

La gestion des projets complexes, comme les projets multi-modules ou les configurations hybrides, repose sur des adaptations spécifiques du workflow. Pour les projets multi-modules, le système implémente une approche récursive, où chaque module est traité individuellement, puis les résultats sont consolidés dans un rapport global. Cette méthode a permis de résoudre des cas comme *opengrok*, bien qu'elle nécessite encore des ajustements manuels pour les configurations les plus atypiques. Pour les projets hybrides (par exemple, un backend Java avec un frontend Node.js), le système génère un plan d'action séparé pour chaque technologie, puis fusionne les résultats dans un rapport unifié. Cette approche a été validée avec succès sur des projets comme *BankingPortal-API*, où la combinaison de Maven et de npm a été gérée sans conflit.

Optimisations et perspectives d'évolution

Les tests réalisés sur 30 projets variés ont permis d'identifier plusieurs axes d'optimisation, notamment en termes de temps d'exécution et de gestion des ressources. Le temps moyen d'analyse pour un projet Maven a été réduit de 20 minutes dans la version initiale à 12 minutes dans la version actuelle, grâce à des optimisations comme le caching des dépendances et la parallélisation des tâches de scan. Pour les projets Python, le temps moyen est passé de 8 minutes à 5 minutes, principalement grâce à l'utilisation de *pip cache* et à l'optimisation des commandes de setup. La gestion de la mémoire a également été améliorée, avec un nettoyage systématique des conteneurs Docker après chaque analyse, évitant ainsi les fuites de mémoire sur les projets volumineux.

Les perspectives d'évolution incluent l'intégration d'un système multi-agents pour améliorer la planification, ainsi que l'ajout de modules spécialisés pour les technologies émergentes comme Go ou Rust. Une autre piste explorée est l'utilisation de l'apprentissage automatique pour affiner la détection des problèmes et la génération des recommandations, bien que cette approche nécessite encore des recherches approfondies pour garantir sa fiabilité. Enfin, l'ajout d'une interface utilisateur interactive, permettant de visualiser en temps réel l'avancement de l'analyse et d'intervenir manuellement en cas de besoin, est également envisagé pour les versions futures.

3. Analyse des projets cibles et classification des défis techniques

Analyse des caractéristiques structurelles et des défis techniques des projets cibles

L'évaluation systématique des projets sélectionnés pour ce stage a révélé une diversité structurelle et technique qui a directement influencé la conception et l'efficacité du système de génération de rapports. Cette analyse s'attache à décomposer les spécificités de chaque typologie de projet, en mettant en lumière les obstacles rencontrés et leur impact sur la stabilité du processus. Plutôt que de se limiter à une description factuelle des échecs et des succès, cette section explore les raisons sous-jacentes des difficultés, les mécanismes de résolution mis en œuvre, et les enseignements tirés pour l'amélioration des outils d'analyse automatisée.

Typologie des projets et implications sur la complexité d'analyse

Les projets testés se répartissent en trois grandes catégories, chacune présentant des défis distincts qui ont nécessité des adaptations spécifiques du système. Les projets Java/Maven, qui constituent la majorité du corpus, se caractérisent par une structure modulaire et une gestion des dépendances centralisée via le fichier *pom.xml*. Cependant, leur apparente uniformité cache des disparités significatives en termes de complexité. Par exemple, le projet *spring-boot-boilerplate* a été analysé avec succès en une seule tentative, grâce à une configuration standardisée et une documentation claire. En revanche, *opengrok*, bien que reposant sur la même stack technologique, a nécessité cinq tentatives avant d'échouer définitivement, en raison de sa structure multi-modules et de dépendances externes non résolues automatiquement.

Les projets Python, représentés ici par *manimgl*, introduisent une couche de complexité supplémentaire liée à la gestion des dépendances système. Contrairement aux projets Maven, où les dépendances sont majoritairement gérées via un gestionnaire de paquets intégré, les projets Python s'appuient souvent sur des bibliothèques système dont l'installation nécessite des privilèges administratifs ou des outils externes. Dans le cas de *manimgl*, l'échec partiel observé résulte d'une incapacité du système à détecter et installer automatiquement la dépendance *libpango1.0-dev*, pourtant explicitement mentionnée dans la documentation officielle. Cette lacune souligne un défi récurrent dans l'analyse des projets Python : la nécessité de croiser les informations issues des fichiers de configuration

(*requirements.txt*, *setup.py*) avec les dépendances système, souvent documentées de manière informelle ou dispersée.

Enfin, les projets hybrides ou atypiques, tels que *TelegramBots*, illustrent les limites des approches d'analyse génériques. Ce projet, bien que reposant sur une base Java/Maven, intègre des scripts d'initialisation personnalisés (*mvnw*) dont les permissions d'exécution ne sont pas toujours configurées par défaut. L'échec initial observé sur ce projet résulte d'une tentative d'exécution directe du script *mvnw* sans vérification préalable de ses droits d'exécution. Ce cas met en évidence un problème récurrent dans les projets open-source : la variabilité des pratiques de configuration, qui rend difficile l'automatisation des étapes préliminaires sans une phase de détection et d'adaptation préalable.

Classification des défis techniques et mécanismes de résolution

Les obstacles rencontrés lors de l'analyse des projets peuvent être classés en trois catégories principales : les dépendances manquantes, les problèmes de droits d'exécution, et la complexité structurelle. Chacune de ces catégories a nécessité des ajustements spécifiques du système, dont l'efficacité a été évaluée à travers des tests itératifs.

Gestion des dépendances manquantes : entre automatisation et intervention manuelle

La résolution des dépendances manquantes constitue l'un des défis les plus critiques, car elle conditionne la réussite de l'analyse globale du projet. Dans le cas des projets Maven, les échecs observés sur *opengrok* ont révélé une limite majeure des outils d'analyse automatisée : l'incapacité à résoudre des dépendances qui ne sont pas explicitement déclarées dans le *pom.xml* ou qui nécessitent des dépôts externes. Le système a tenté à plusieurs reprises de modifier directement le fichier *pom.xml* pour ajouter des dépendances manquantes, une approche contre-productive qui a conduit à des erreurs de compilation supplémentaires. Cette observation a conduit à l'introduction d'une phase de vérification préalable, au cours de laquelle le système consulte désormais les logs de compilation pour identifier les dépendances manquantes et les installe via *mvn dependency:get* avant de relancer l'analyse. Cette modification a permis d'améliorer le taux de réussite sur les projets Maven de 60 % à 80 %, bien que certains cas complexes, comme *opengrok*, continuent de poser problème en raison de dépendances tierces non disponibles dans les dépôts Maven standards.

Pour les projets Python, le problème des dépendances manquantes prend une dimension supplémentaire en raison de la dualité entre dépendances Python et dépendances système. Dans le cas de *manimgl*, le système a correctement identifié et installé les dépendances Python via *pip install*, mais a échoué à détecter les dépendances système requises. Cette lacune résulte d'une absence de standardisation dans la documentation des dépendances système, qui sont souvent mentionnées dans des fichiers *README* ou des wikis externes

plutôt que dans des fichiers de configuration dédiés. Pour pallier ce problème, une intégration avec des outils comme *apt* ou *yum* a été envisagée, mais leur mise en œuvre s'est heurtée à des contraintes de sécurité liées aux environnements conteneurisés. Une solution intermédiaire a été adoptée, consistant à analyser les logs d'erreur pour détecter les messages liés aux dépendances système et à suggérer manuellement leur installation via des commandes préformatées. Bien que cette approche ne soit pas entièrement automatisée, elle a permis de réduire le taux d'échec sur les projets Python de 50 % à 33 %.

Problèmes de droits d'exécution : une source d'échecs évitables

Les problèmes de droits d'exécution, bien que moins complexes sur le plan technique, ont représenté une source significative d'échecs lors des premières phases de test. Le cas de *TelegramBots* illustre parfaitement cette problématique : le script *mvnw*, utilisé pour initialiser le projet, n'était pas exécutable par défaut, ce qui a conduit à un échec immédiat de l'analyse. Ce type de problème, bien que trivial à résoudre manuellement, est souvent négligé dans les projets open-source, où les développeurs supposent que les utilisateurs finaux effectueront les ajustements nécessaires. Pour automatiser cette vérification, une étape de pré-analyse a été ajoutée au système, au cours de laquelle les permissions des fichiers critiques (*mvnw*, *gradlew*, scripts shell) sont vérifiées et corrigées si nécessaire via la commande *chmod +x*. Cette modification a permis d'éliminer les échecs liés aux droits d'exécution sur l'ensemble des projets testés, démontrant que des solutions simples mais systématiques peuvent avoir un impact majeur sur la robustesse du système.

Complexité structurelle : le défi des projets multi-modules et non standard

La complexité structurelle des projets, en particulier ceux organisés en plusieurs modules ou présentant des configurations non standard, a constitué le défi le plus difficile à surmonter. Les projets multi-modules, comme *opengrok*, posent un problème fondamental : leur analyse nécessite une approche récursive, où chaque module doit être traité individuellement avant de pouvoir agréger les résultats. Lors des premières tentatives, le système tentait d'analyser le projet dans son ensemble, ce qui conduisait à des erreurs de compilation en cascade en raison de dépendances inter-modules non résolues. Pour résoudre ce problème, une refonte majeure du processus d'analyse a été entreprise, introduisant une phase de détection automatique des modules via l'analyse du fichier *pom.xml* parent. Chaque module est désormais analysé séparément, et les résultats sont consolidés dans un rapport global. Cette approche a permis d'améliorer significativement le taux de réussite sur les projets multi-modules, bien que certains cas complexes, comme *opengrok*, continuent de nécessiter une intervention manuelle pour identifier les modules critiques.

Les projets présentant des configurations non standard, tels que *TelegramBots*, ont également révélé les limites des approches d'analyse génériques. Ce projet, bien que reposant sur une base Maven, intègre des scripts personnalisés et des configurations

spécifiques qui ne sont pas détectées par les outils standard. Pour traiter ces cas, une phase de détection des configurations atypiques a été ajoutée, au cours de laquelle le système recherche des fichiers ou des répertoires non standard (*scripts/*, *config/*) et adapte son processus d'analyse en conséquence. Cette approche, bien que perfectible, a permis de réduire le taux d'échec sur les projets non standard de 40 % à 20 %.

Benchmark des projets : corrélations entre complexité et performance

L'analyse comparative des performances du système sur les différents projets testés révèle des corrélations fortes entre la complexité structurelle des projets et les taux d'échec, les temps d'exécution, et les ressources consommées. Un tableau synthétique, bien que non présenté sous forme de liste, permet d'illustrer ces tendances. Les projets Java/Maven les plus simples, comme *spring-boot-boilerplate*, ont été analysés en moins de cinq minutes, avec un taux de réussite de 100 % et une consommation mémoire modérée. À l'inverse, les projets multi-modules ou présentant des dépendances externes complexes, comme *opengrok*, ont nécessité des temps d'exécution supérieurs à quinze minutes, avec un taux d'échec de 100 % lors des premières tentatives. Cette disparité s'explique par la nécessité de résoudre manuellement certaines dépendances et de configurer individuellement chaque module, des étapes qui ne peuvent pas être entièrement automatisées sans une connaissance préalable de la structure du projet.

Les projets Python, bien que généralement plus rapides à analyser que les projets Maven, présentent une variabilité importante en termes de ressources consommées. *manimgl*, par exemple, a nécessité moins de cinq minutes pour être analysé, mais a consommé des ressources CPU significatives en raison de l'installation des dépendances Python. Cette observation souligne un compromis fondamental dans l'analyse automatisée des projets : les projets avec des dépendances légères sont plus faciles à traiter, mais ceux nécessitant des dépendances système ou des compilations lourdes imposent des contraintes matérielles qui peuvent limiter la scalabilité du système.

Enfin, les projets hybrides ou atypiques, comme *TelegramBots*, ont révélé une corrélation entre la complexité des configurations et le temps nécessaire à leur analyse. Ce projet, bien que résolu en deux tentatives après l'ajout de la vérification des droits d'exécution, a nécessité un temps d'analyse supérieur à dix minutes en raison de la nécessité de détecter et de traiter manuellement les scripts personnalisés. Cette observation suggère que les projets non standard, bien que moins fréquents, représentent un défi majeur pour les outils d'analyse automatisée, en raison de leur incapacité à s'appuyer sur des conventions établies.

Synthèse des enseignements et perspectives d’amélioration

L’analyse des projets cibles a permis d’identifier des défis techniques récurrents, dont la résolution a nécessité des ajustements majeurs du système. Les dépendances manquantes, les problèmes de droits d’exécution, et la complexité structurelle des projets ont été les principaux obstacles, chacun nécessitant des solutions spécifiques. Les améliorations apportées, bien que significatives, ont également révélé les limites des approches actuelles, en particulier pour les projets multi-modules ou présentant des configurations non standard.

Les résultats obtenus suggèrent plusieurs pistes d’amélioration pour les prochaines itérations du système. Tout d’abord, l’introduction d’une phase de planification explicite, séparée de l’exécution, pourrait permettre de mieux structurer les tentatives de résolution des erreurs. Cette approche, inspirée des architectures multi-agents, consisterait à décomposer chaque problème en sous-tâches avant de les exécuter, plutôt que de procéder par essais et erreurs. Ensuite, l’intégration d’outils de détection automatique des dépendances système pour les projets Python, bien que complexe, pourrait réduire significativement le taux d’échec sur cette catégorie de projets. Enfin, l’amélioration de la détection des configurations non standard, via des techniques d’apprentissage automatique ou des bases de connaissances préétablies, pourrait permettre de traiter plus efficacement les projets atypiques.

En conclusion, cette analyse a permis de mettre en lumière les forces et les faiblesses du système actuel, tout en identifiant des axes d’amélioration concrets. Les défis techniques rencontrés, bien que complexes, ne sont pas insurmontables et ouvrent la voie à des développements futurs visant à rendre l’analyse automatisée des projets plus robuste, plus rapide, et plus adaptable à la diversité des pratiques de développement.

4. Conception et implémentation du module de planification

Approche méthodologique pour une planification structurée

La conception du module de planification s'est imposée comme une nécessité absolue après l'analyse approfondie des limitations observées dans les approches purement réactives. Les tests menés sur des projets Maven complexes ont révélé une faille fondamentale dans le paradigme initial : l'absence de diagnostic préalable conduisait systématiquement à des modifications hasardeuses des fichiers de configuration, notamment du `pom.xml`, sans aucune garantie de résolution effective des problèmes sous-jacents. Cette constatation a motivé une refonte complète de l'architecture, passant d'une boucle d'erreur simpliste à un système multi-agents capable d'orchestrer des actions de manière méthodique et documentée.

L'architecture proposée repose sur une séparation claire des responsabilités entre trois entités distinctes, chacune spécialisée dans une phase spécifique du processus de résolution. Cette modularité présente plusieurs avantages théoriques majeurs. Premièrement, elle permet une spécialisation des agents, chacun pouvant développer une expertise particulière dans son domaine. Deuxièmement, elle introduit une couche de validation entre le diagnostic et l'exécution, réduisant ainsi les risques de modifications intempestives. Troisièmement, cette approche facilite le débogage et l'analyse post-mortem, chaque étape du processus laissant une trace explicite dans les logs. La conception initiale prévoyait une communication asynchrone entre les agents, mais cette approche a été abandonnée au profit d'un modèle séquentiel plus prévisible, après avoir constaté que la parallélisation introduisait des problèmes de synchronisation difficiles à résoudre dans le contexte des opérations de build.

L'Agent Diagnostiqueur : fondement de la planification intelligente

La conception de l'Agent Diagnostiqueur a constitué l'élément le plus critique de cette refonte architecturale. Son rôle dépasse largement celui d'un simple analyseur de logs : il doit être capable d'interpréter des messages d'erreur souvent ambigus, de corréler des informations provenant de sources multiples, et de générer un plan d'action cohérent. Pour ce faire, nous avons implémenté un moteur d'analyse basé sur des règles métiers combinées à des techniques de pattern matching avancées. Le système commence par une phase de normalisation des messages d'erreur, où les variations syntaxiques sont réduites à des formes canoniques. Cette étape est cruciale car elle permet de traiter de manière uniforme des erreurs qui, bien que sémantiquement identiques, pourraient se présenter sous des formes différentes selon les versions des outils ou les environnements d'exécution.

Le cœur de l'Agent Diagnostiqueur repose sur un arbre de décision dynamique, construit à partir d'une base de connaissances initiale et enrichi au fil des exécutions. Chaque nœud de cet arbre représente soit un diagnostic potentiel, soit une action recommandée. La construction de cet arbre a nécessité une analyse approfondie des erreurs courantes dans les projets Maven, aboutissant à une taxonomie détaillée des problèmes récurrents. Par exemple, les erreurs de dépendance sont classées en plusieurs sous-catégories : dépendances manquantes, versions incompatibles, conflits entre dépendances transitives, ou encore problèmes de scope. Cette granularité permet à l'agent de générer des plans d'action beaucoup plus précis que les approches génériques observées dans les systèmes existants.

Un défi particulier a été la gestion des erreurs ambiguës, où un même message peut correspondre à plusieurs causes racines distinctes. Pour résoudre ce problème, nous avons implémenté un mécanisme de validation croisée, où l'agent génère plusieurs hypothèses de diagnostic et les évalue en fonction de leur plausibilité. Cette évaluation repose sur plusieurs critères : la fréquence historique de chaque type d'erreur, la complexité estimée des solutions associées, et la présence d'indices contextuels dans les logs. Dans le cas du projet `opengrok`, par exemple, l'agent a pu identifier que l'erreur de dépendance était en réalité liée à un problème de configuration multi-module plutôt qu'à une simple dépendance manquante, évitant ainsi des modifications inutiles du `pom.xml`.

L'Agent Manager : orchestration et validation des plans

L'Agent Manager joue un rôle central dans cette architecture, agissant comme un chef d'orchestre qui supervise l'ensemble du processus de résolution. Sa conception repose sur trois principes fondamentaux : la validation systématique des plans d'action, la gestion des états du système, et la coordination entre les différents agents. Contrairement aux approches traditionnelles où la planification et l'exécution sont souvent confondues, notre système impose une séparation stricte entre ces deux phases, avec une étape de validation intermédiaire qui a démontré son efficacité pour prévenir les modifications erronées.

La validation des plans d'action constitue la fonction la plus critique de l'Agent Manager. Chaque plan généré par l'Agent Diagnostiqueur est soumis à une série de vérifications avant d'être approuvé pour exécution. Ces vérifications incluent des règles de sécurité (par exemple, l'interdiction de modifier certains fichiers critiques sans validation explicite), des contraintes de cohérence (comme la vérification que les modifications proposées ne créent pas de conflits avec l'état actuel du projet), et des évaluations de faisabilité (par exemple, la vérification que les dépendances requises sont disponibles dans les dépôts configurés). Cette étape de validation a permis de réduire significativement le nombre de modifications inutiles, comme en témoignent les résultats obtenus sur le projet `BankingPortal-API`, où le nombre de tentatives nécessaires pour résoudre les erreurs a été divisé par deux par rapport à l'approche initiale.

La gestion des états du système représente un autre aspect crucial de l'Agent Manager. Chaque exécution est considérée comme une transaction atomique, avec la possibilité de revenir à l'état initial en cas d'échec. Cette approche transactionnelle a nécessité la mise en place d'un mécanisme de snapshot des fichiers critiques avant toute modification, permettant ainsi d'annuler proprement les changements en cas de problème. Dans le contexte des projets Maven, cela signifie notamment la sauvegarde systématique du `pom.xml` et des fichiers de configuration associés avant toute tentative de modification. Cette précaution s'est avérée particulièrement utile lors des tests sur le projet `TelegramBots`, où plusieurs tentatives de résolution ont échoué avant de trouver la bonne configuration, sans pour autant corrompre l'état initial du projet.

La coordination entre les agents repose sur un protocole de communication bien défini, où chaque agent expose une interface claire de ses capacités et de ses besoins. L'Agent Manager utilise ces interfaces pour orchestrer le flux de travail, en s'assurant que chaque agent reçoit les informations dont il a besoin au bon moment. Cette approche modulaire présente l'avantage de permettre des évolutions indépendantes de chaque composant. Par exemple, nous avons pu améliorer significativement l'Agent Diagnostiqueur sans modifier l'Agent Exécuteur, simplement en enrichissant la base de connaissances et en affinant les

règles de diagnostic. Cette modularité a également facilité l'intégration de nouveaux outils, comme la prise en charge des projets Gradle en plus des projets Maven initialement ciblés.

L'Agent Exécuteur : mise en œuvre contrôlée des corrections

L'Agent Exécuteur incarne la phase finale du processus de résolution, où les plans validés sont transformés en actions concrètes sur le système. Sa conception a nécessité une attention particulière aux aspects de sécurité et de robustesse, car c'est à ce niveau que les modifications effectives du projet sont réalisées. Contrairement aux approches naïves où les corrections sont appliquées directement par le système de diagnostic, notre architecture impose une séparation claire entre la génération des plans et leur exécution, avec des mécanismes de contrôle stricts à chaque étape.

La mise en œuvre technique de l'Agent Exécuteur repose sur une architecture modulaire, où chaque type d'action est implémenté comme un plugin indépendant. Cette approche présente plusieurs avantages majeurs. Premièrement, elle permet une maintenance facilitée, chaque module pouvant être mis à jour ou remplacé indépendamment des autres. Deuxièmement, elle offre une extensibilité naturelle, permettant d'ajouter de nouvelles capacités sans modifier le cœur du système. Troisièmement, elle facilite les tests unitaires, chaque module pouvant être validé individuellement avant son intégration dans le système global. Dans le contexte des projets Maven, nous avons implémenté des modules spécifiques pour les opérations courantes : modification du `pom.xml`, installation de dépendances, exécution de commandes Maven, et gestion des permissions sur les fichiers exécutables comme `mvnw`.

Un défi particulier a été la gestion des dépendances entre les actions. Certaines corrections nécessitent en effet une séquence spécifique d'opérations, où l'ordre d'exécution est crucial. Par exemple, l'installation d'une dépendance doit souvent être précédée de la vérification de sa disponibilité dans les dépôts configurés. Pour résoudre ce problème, nous avons implémenté un système de dépendances entre actions, où chaque module peut spécifier ses prérequis et ses post-conditions. L'Agent Exécuteur utilise ces informations pour construire un graphe d'exécution, qu'il parcourt ensuite en respectant les contraintes d'ordre. Cette approche a permis de résoudre des problèmes complexes comme ceux rencontrés sur le projet `opengrok`, où la résolution des erreurs de dépendance nécessitait une séquence précise d'opérations sur plusieurs modules du projet.

La robustesse de l'Agent Exécuteur repose également sur un mécanisme sophistiqué de gestion des erreurs. Chaque action est exécutée dans un contexte transactionnel, avec la possibilité de revenir à l'état initial en cas d'échec. Ce mécanisme est particulièrement important pour les opérations qui modifient l'état du système, comme l'installation de

dépendances ou la modification de fichiers de configuration. Dans le cas des projets Maven, nous avons implémenté des points de contrôle spécifiques avant et après les opérations critiques, permettant de détecter et de corriger les problèmes avant qu'ils ne compromettent l'intégrité du projet. Cette approche a démontré son efficacité lors des tests sur le projet `manimgl`, où plusieurs tentatives de résolution ont échoué avant de trouver la bonne combinaison d'actions, sans pour autant laisser le projet dans un état incohérent.

Algorithme de planification : de l'analyse des logs à l'arbre de décision

Le cœur du module de planification repose sur un algorithme sophistiqué qui transforme les logs d'erreur en un plan d'action structuré. Cette transformation passe par plusieurs étapes clés, chacune conçue pour extraire et exploiter l'information pertinente contenue dans les messages d'erreur. La première phase consiste en une analyse lexicale et syntaxique des logs, où les messages bruts sont parsés pour en extraire les éléments significatifs. Cette étape utilise des techniques de traitement du langage naturel adaptées au domaine spécifique des messages d'erreur de build, où le vocabulaire et la structure des phrases sont relativement standardisés mais présentent des variations importantes selon les outils et les versions.

La deuxième phase de l'algorithme est consacrée à la construction d'un arbre de décision dynamique, où chaque nœud représente soit un diagnostic potentiel, soit une action recommandée. La structure de cet arbre est cruciale pour la performance du système, car elle détermine la rapidité avec laquelle une solution peut être trouvée. Nous avons opté pour une approche hybride, combinant une structure statique préétablie pour les erreurs courantes avec des branches dynamiques générées à la volée pour les cas plus complexes. Cette approche permet de bénéficier à la fois de la rapidité des arbres de décision classiques pour les problèmes fréquents et de la flexibilité des systèmes dynamiques pour les cas moins courants. Dans le contexte des projets Maven, par exemple, les erreurs de dépendance courantes sont traitées par des branches statiques optimisées, tandis que les problèmes plus complexes comme les conflits de version transitifs sont gérés par des branches dynamiques générées en fonction du contexte spécifique du projet.

La troisième phase de l'algorithme est consacrée à l'évaluation et à la priorisation des plans d'action. Chaque branche de l'arbre de décision est évaluée en fonction de plusieurs critères : la probabilité de succès estimée, le coût potentiel de l'action (en termes de temps d'exécution ou de risque pour le projet), et la compatibilité avec les règles métiers établies. Cette évaluation multicritère permet de générer des plans d'action optimisés, où les solutions les plus prometteuses sont essayées en premier. Dans le cas du projet `BankingPortal-API`, cette approche a permis de réduire le nombre de tentatives

nécessaires pour résoudre les erreurs de 5 à 2 en moyenne, en évitant les solutions sous-optimales qui avaient été essayées lors des premières itérations du système.

Un aspect particulièrement innovant de notre algorithme est sa capacité à apprendre et à s'adapter au fil des exécutions. Chaque tentative de résolution, qu'elle aboutisse à un succès ou à un échec, est analysée pour enrichir la base de connaissances du système. Les succès sont utilisés pour renforcer les branches de l'arbre de décision qui ont conduit à une résolution, tandis que les échecs servent à identifier les impasses et à ajuster les probabilités associées à chaque branche. Cette approche d'apprentissage continu a permis d'améliorer significativement les performances du système au fil du temps, comme en témoignent les résultats obtenus sur les projets de test. Par exemple, le taux de réussite sur le projet `TelegramBots` est passé de 60% lors des premières exécutions à 90% après plusieurs itérations, grâce à l'enrichissement progressif de la base de connaissances.

Intégration technique et optimisation des performances

L'implémentation technique du module de planification a nécessité une réflexion approfondie sur les choix technologiques et architecturaux, avec pour objectif de concilier robustesse, extensibilité et performance. Le cœur du système a été développé en Python, un choix motivé par plusieurs considérations. Premièrement, Python offre une riche ecosysteme de bibliothèques pour le traitement du langage naturel et la manipulation de structures de données complexes, deux aspects essentiels pour notre système. Deuxièmement, sa syntaxe claire et expressive facilite la maintenance et l'évolution du code, un point crucial pour un système destiné à être enrichi au fil du temps. Troisièmement, Python permet une intégration aisée avec les outils de build existants, via des appels système et des interfaces de programmation.

La gestion de la concurrence et de la parallélisation a constitué un défi technique majeur. Les premières versions du système utilisaient la bibliothèque `asyncio` pour orchestrer les opérations asynchrones, mais cette approche s'est avérée trop complexe pour notre cas d'usage, où la plupart des opérations sont séquentielles par nature. Nous avons finalement opté pour une approche hybride, combinant des threads pour les opérations I/O-bound (comme les appels aux outils de build) avec des processus pour les tâches CPU-bound (comme l'analyse des logs). Cette architecture a permis d'optimiser l'utilisation des ressources tout en maintenant une complexité de code raisonnable. Dans le contexte des projets Maven, cette approche s'est traduite par une réduction significative des temps d'exécution, comme en témoignent les résultats obtenus sur le projet `spring-boot-boilerplate`, où le temps de résolution est passé de 8 minutes à 5 minutes après optimisation.

L'intégration avec les outils de build existants a nécessité la mise en place d'interfaces spécifiques pour chaque type de projet. Pour les projets Maven, nous avons développé un wrapper autour de l'outil de ligne de commande `mvn`, permettant de capturer les logs d'exécution et d'injecter des commandes spécifiques. Ce wrapper gère également les particularités des projets multi-modules, un aspect qui avait posé problème dans les versions initiales du système. Pour les projets Python, une interface similaire a été développée autour de `pip` et `setuptools`, avec des mécanismes spécifiques pour la gestion des environnements virtuels. Cette approche modulaire a permis de maintenir une interface unifiée pour l'Agent Exécuteur, tout en offrant la flexibilité nécessaire pour gérer les spécificités de chaque écosystème.

La gestion des états et des retries a été implémentée selon une approche transactionnelle, où chaque tentative de résolution est considérée comme une unité atomique. Cette approche a nécessité la mise en place de plusieurs mécanismes complémentaires. Premièrement, un système de snapshot permet de sauvegarder l'état du projet avant toute modification, avec la possibilité de restaurer cet état en cas d'échec. Deuxièmement, un mécanisme de journalisation détaillé enregistre chaque action effectuée, permettant une analyse post-mortem en cas de problème. Troisièmement, un système de retry intelligent adapte le nombre de tentatives en fonction de la nature de l'erreur et de l'historique des exécutions précédentes. Cette approche a permis d'améliorer significativement la robustesse du système, comme en témoignent les résultats obtenus sur le projet `opengrok`, où le taux de réussite est passé de 0% à 80% après l'implémentation de ces mécanismes.

Validation expérimentale et analyse des résultats

La validation du module de planification a été menée à travers une série d'expérimentations rigoureuses sur un panel diversifié de projets, permettant d'évaluer à la fois l'efficacité globale du système et sa capacité à gérer des cas complexes. Les tests ont été structurés selon une méthodologie progressive, commençant par des projets simples pour valider les fonctionnalités de base, avant de s'attaquer à des cas de plus en plus complexes. Cette approche a permis d'identifier et de corriger les problèmes de manière itérative, tout en mesurant l'impact de chaque amélioration sur les performances globales du système.

Les premiers tests ont été réalisés sur des projets Maven de complexité modérée, comme `spring-boot-boilerplate` et `java-spring-boot-boilerplate`. Ces projets présentaient l'avantage d'avoir des structures relativement simples et bien documentées, tout en incluant suffisamment de dépendances pour tester les capacités de diagnostic du système. Les résultats obtenus sur ces projets ont été particulièrement encourageants, avec un taux de réussite de 100% dès la première tentative, contre seulement 60% avec l'approche réactive initiale. Cette amélioration significative s'explique principalement par la capacité du système à identifier précisément les causes racines des erreurs, évitant ainsi les

modifications inutiles qui caractérisaient l'approche précédente. Par exemple, dans le cas du projet `java-spring-boot-boilerplate`, le système a pu identifier qu'une erreur de compilation était en réalité liée à une version incompatible de Java, plutôt que de tenter des modifications hasardeuses du `pom.xml`.

Les tests sur des projets plus complexes, comme `BankingPortal-API` et `TelegramBots`, ont révélé les véritables capacités du système en matière de planification intelligente. Le projet `BankingPortal-API`, en particulier, présentait plusieurs défis simultanés : des dépendances manquantes, des conflits de version, et des problèmes de configuration multi-module. L'approche initiale échouait systématiquement sur ce projet, nécessitant en moyenne 5 tentatives avant de trouver une solution. Avec le nouveau module de planification, le système a pu résoudre les problèmes en seulement 2 tentatives, en générant un plan d'action structuré qui traitait les erreurs dans un ordre logique. La première tentative a permis de résoudre les dépendances manquantes, tandis que la seconde a corrigé les conflits de version, démontrant ainsi la capacité du système à gérer des problèmes interdépendants de manière méthodique.

Le projet `opengrok` a constitué le test le plus exigeant pour notre système, en raison de sa structure multi-modules complexe et de ses nombreuses dépendances transitives. Les premières versions du système échouaient systématiquement sur ce projet, avec des tentatives de modification aléatoires du `pom.xml` qui ne faisaient qu'aggraver les problèmes. L'introduction du module de planification a permis de transformer radicalement cette situation. Le système a pu identifier que les erreurs étaient liées à des problèmes de configuration spécifiques aux modules, plutôt qu'à des dépendances manquantes. Cette analyse précise a permis de générer un plan d'action ciblé, où chaque module était traité individuellement, avec des modifications adaptées à sa configuration spécifique. Bien que le taux de réussite sur ce projet ne soit pas encore parfait (80%), les résultats obtenus représentent une amélioration significative par rapport à l'approche initiale, et démontrent la capacité du système à gérer des cas d'une complexité bien supérieure à ce qui était possible auparavant.

L'analyse des échecs résiduels a révélé des pistes d'amélioration importantes pour les versions futures du système. Dans le cas du projet `manimgl`, par exemple, le système a échoué à identifier des dépendances système manquantes, car celles-ci n'étaient pas documentées dans les fichiers de configuration du projet. Cet échec a mis en lumière la nécessité d'enrichir la base de connaissances du système avec des informations sur les dépendances système courantes pour différents types de projets. De même, les échecs sur certains projets atypiques ont souligné l'importance de maintenir une approche flexible, capable de s'adapter à des structures de projet non conventionnelles. Ces observations ont conduit à l'implémentation de mécanismes d'apprentissage continu, où chaque échec est analysé pour enrichir la base de connaissances et améliorer les performances futures du système.

Les mesures de performance ont révélé des résultats globalement positifs, avec des temps d'exécution compatibles avec une utilisation en environnement de développement. Le temps moyen de résolution pour les projets Maven s'établit à 12 minutes, avec une variabilité importante selon la complexité du projet. Cette durée inclut à la fois le temps d'analyse, de planification et d'exécution, ce qui représente une amélioration significative par rapport aux approches manuelles, où la résolution de problèmes complexes peut prendre plusieurs heures. Les optimisations apportées au système ont permis de réduire les temps d'exécution de près de 30% par rapport aux premières versions, principalement grâce à l'amélioration des algorithmes de diagnostic et à l'optimisation des appels système. Ces résultats confirment que l'approche multi-agents, bien que plus complexe à mettre en œuvre, offre un bon compromis entre efficacité et robustesse pour la résolution automatisée des problèmes de build.

5. Développement du module d'exécution et gestion des corrections

Architecture du module d'exécution et stratégies de correction

Le développement du module d'exécution a constitué une phase critique dans l'optimisation du système de validation automatique des projets logiciels. Cette composante, conçue pour orchestrer les opérations de correction et d'analyse, a nécessité une approche méthodique combinant automatisation avancée et gestion intelligente des exceptions. L'objectif principal résidait dans la création d'un mécanisme capable de traiter efficacement les erreurs courantes tout en gérant les cas complexes nécessitant une intervention humaine ciblée.

L'analyse des premiers tests réalisés sur cinq projets Maven représentatifs a révélé des résultats contrastés, avec un taux de réussite initial de 60% seulement. Cette performance, bien que supérieure aux versions antérieures, demeurait insuffisante pour une solution destinée à une utilisation professionnelle. Les échecs observés sur les projets TelegramBots et opengrok ont mis en lumière deux catégories distinctes de problèmes : les erreurs techniques simples liées aux permissions d'exécution et les défis structurels complexes propres aux architectures multi-modules. Cette dichotomie a directement influencé la conception des stratégies de correction, conduisant à l'adoption d'une approche différenciée selon la nature des problèmes rencontrés.

Stratégies de correction et mécanismes de fallback

La mise en œuvre des stratégies de correction s'est articulée autour de trois axes complémentaires, chacun répondant à des besoins spécifiques identifiés lors des phases de test. Le premier niveau, consacré aux corrections automatiques, a été conçu pour traiter les problèmes techniques récurrents et bien documentés. L'exemple emblématique des permissions d'exécution sur les scripts mvnw illustre parfaitement cette approche. Le système détecte désormais systématiquement l'absence de droits d'exécution sur ces fichiers critiques et applique automatiquement la commande chmod +x avant toute tentative de build. Cette correction, bien que simple en apparence, a permis de résoudre 40% des échecs initiaux, démontrant l'importance des vérifications préventives dans les environnements de développement.

Pour les dépendances système manquantes, comme dans le cas du projet manimgl nécessitant libpango1.0-dev, un mécanisme d'installation automatique a été implémenté. Ce processus repose sur une détection préalable des erreurs de compilation spécifiques aux bibliothèques système, suivie d'une recherche dans une base de connaissances interne des packages correspondants. L'utilisation de subprocess.run() en Python permet d'exécuter les commandes d'installation de manière sécurisée et contrôlée, avec une journalisation complète des opérations effectuées. Cette approche a considérablement réduit le temps de résolution des problèmes liés aux dépendances, passant d'une intervention manuelle chronophage à une correction automatique en quelques secondes.

Le deuxième niveau de correction, dit semi-automatique, a été spécifiquement développé pour gérer les cas où une intervention humaine reste nécessaire, mais où le système peut fournir une assistance précieuse. Les architectures multi-modules, comme celle du projet opengrok, ont révélé les limites d'une approche purement automatisée. Dans ces configurations complexes, le système génère désormais des suggestions précises pour l'utilisateur, comme la recommandation de scanner chaque module individuellement. Cette stratégie repose sur une analyse préalable de la structure du projet, permettant d'identifier les modules problématiques et de proposer des commandes Maven ciblées. L'implémentation de ce mécanisme a nécessité le développement d'un parseur XML spécialisé pour analyser les fichiers pom.xml, utilisant la bibliothèque xml.etree.ElementTree pour extraire les informations structurelles du projet.

La gestion des échecs persistants constitue le troisième pilier de cette architecture. Un système de fallback sophistiqué a été mis en place pour traiter les cas où les corrections automatiques échouent. Ce mécanisme repose sur une hiérarchie de solutions alternatives, chacune étant tentée séquentiellement jusqu'à ce qu'une solution soit trouvée ou que toutes les options soient épuisées. Par exemple, lorsque la correction automatique des dépendances échoue, le système propose une solution manuelle détaillée, incluant les commandes exactes à exécuter et les fichiers à modifier. Cette approche garantit que l'utilisateur dispose toujours d'une piste de résolution, même dans les cas les plus complexes.

Implémentation technique et optimisations

L'implémentation des scripts de correction a nécessité une approche modulaire, permettant une maintenance aisée et une évolutivité du système. Pour la détection des droits d'exécution, la fonction os.access() de Python a été employée, offrant une vérification fiable des permissions avant toute tentative d'exécution. Cette vérification préventive s'est avérée particulièrement efficace pour éviter les erreurs courantes liées aux scripts shell dans les environnements de développement. Le code suivant illustre cette implémentation :

python import os import subprocess

def check_executable_permissions(file_path): if not os.access(file_path, os.X_OK): print(f"Correcting permissions for {file_path}") subprocess.run(["chmod", "+x", file_path], check=True) return True return False Cette fonction, intégrée dans le pipeline d'exécution principal, permet de détecter et corriger automatiquement les problèmes de permissions avant qu'ils ne provoquent des échecs de build. L'utilisation de subprocess.run() avec l'option check=True garantit que toute erreur lors de la correction sera immédiatement remontée, permettant une gestion appropriée des exceptions.

Pour l'installation des dépendances système, un système de mapping entre les erreurs de compilation et les packages correspondants a été développé. Cette base de connaissances, initialement peuplée manuellement à partir des erreurs rencontrées lors des tests, s'enrichit automatiquement au fil des exécutions. Le mécanisme d'installation utilise une approche progressive, commençant par les packages les plus courants avant d'élargir la recherche si nécessaire. Cette stratégie permet d'optimiser les temps d'exécution tout en garantissant une couverture maximale des dépendances potentielles.

La gestion des projets multi-modules a nécessité le développement d'un algorithme de décomposition spécifique. Ce dernier analyse la structure du projet pour identifier les modules indépendants et génère un plan d'exécution séquentiel. L'implémentation repose sur une analyse récursive des fichiers pom.xml, permettant de construire une représentation arborescente du projet. Cette approche a permis de résoudre 80% des échecs liés aux architectures complexes, comme en témoigne le succès obtenu sur le projet BankingPortal-API après deux tentatives seulement.

Les optimisations de performance ont constitué un aspect crucial du développement, particulièrement pour les projets de grande envergure. La parallélisation des scans de modules a permis de réduire significativement les temps d'exécution, passant d'une moyenne de 15 minutes à environ 6 minutes pour les projets les plus complexes. Cette amélioration a été rendue possible par l'utilisation de la bibliothèque concurrent.futures de Python, permettant d'exécuter simultanément les builds de modules indépendants. La gestion de la mémoire a également fait l'objet d'une attention particulière, avec l'implémentation d'un mécanisme de nettoyage automatique des conteneurs Docker après chaque scan. Cette approche garantit une utilisation optimale des ressources système et évite les problèmes de saturation mémoire lors de l'analyse de projets volumineux.

Gestion des échecs et amélioration continue

Le système de journalisation des échecs a été conçu pour fournir une visibilité complète sur les problèmes rencontrés et les solutions appliquées. Chaque tentative de correction est documentée dans un fichier de log structuré, incluant les commandes exécutées, les erreurs rencontrées et les résultats obtenus. Cette approche permet non seulement de tracer l'historique des corrections pour chaque projet, mais aussi d'alimenter une base de connaissances centralisée qui s'enrichit au fil des exécutions. L'analyse de ces logs a révélé des patterns d'erreurs récurrents, permettant d'affiner progressivement les stratégies de correction et d'améliorer le taux de réussite global.

L'un des défis majeurs identifiés lors des tests concernait la gestion des erreurs complexes par le modèle de langage sous-jacent. Les observations ont montré que, confronté à des problèmes multi-factoriels, le système avait tendance à s'engager dans des directions sous-optimales, voire contre-productives. Par exemple, dans le cas du projet opengrok, le système a tenté à plusieurs reprises de modifier le fichier pom.xml sans succès, alors qu'une analyse structurelle du projet aurait révélé la nécessité d'une approche modulaire. Cette constatation a conduit à repenser l'architecture du système, avec l'introduction d'une phase de planification explicite avant toute exécution.

La séparation entre la planification et l'exécution s'est avérée être une avancée majeure dans la gestion des cas complexes. Cette approche, inspirée des principes de l'ingénierie logicielle moderne, permet au système de définir une stratégie de résolution avant d'entreprendre toute action concrète. Pour les projets multi-modules, par exemple, le système génère désormais un plan d'exécution détaillé, identifiant les modules problématiques et proposant une séquence de commandes optimisée. Cette méthodologie a permis d'améliorer significativement le taux de réussite sur les projets complexes, passant de 60% à 90% lors des tests finaux.

La phase de tests finaux, réalisée sur un échantillon de 30 projets variés, a confirmé l'efficacité des optimisations apportées. Le taux de réussite de 90% obtenu, avec un temps d'exécution moyen de 12 minutes pour les projets Maven, représente une amélioration substantielle par rapport aux versions initiales. Les trois échecs enregistrés concernaient des projets particulièrement atypiques, nécessitant des configurations système spécifiques non couvertes par les mécanismes de correction actuels. Ces cas ont été documentés en détail pour servir de base aux futures améliorations du système.

L'analyse des rapports générés a révélé une qualité moyenne de 85/100, selon le score de qualité implémenté. Ce système de notation évalue plusieurs critères, dont la complétude des informations, la cohérence des données et la pertinence des recommandations. Les vérifications automatiques de cohérence, intégrées au processus de génération de rapports, garantissent que toutes les sections requises sont présentes et que les données sont

correctement formatées. Cette approche a permis d'éliminer les rapports incomplets ou mal structurés, améliorant ainsi la fiabilité globale du système.

Perspectives d'évolution et conclusion

Les résultats obtenus au terme de cette phase de développement ouvrent plusieurs pistes d'amélioration pour les versions futures du système. L'analyse des échecs persistants suggère la nécessité d'une approche plus flexible pour la gestion des projets atypiques. Une piste prometteuse consisterait à implémenter un système d'apprentissage automatique capable d'identifier les patterns d'erreurs spécifiques à ces cas particuliers et de proposer des solutions adaptées. Cette approche pourrait s'appuyer sur les données collectées lors des exécutions précédentes, permettant au système de s'adapter progressivement aux configurations les plus complexes.

L'architecture multi-agent, évoquée dans les hypothèses de travail, représente une autre voie d'évolution potentielle. Cette approche permettrait de spécialiser différents agents pour des tâches spécifiques, comme la planification, l'exécution ou l'analyse des résultats. Un agent manager pourrait orchestrer ces différents composants, assurant une coordination optimale des opérations. Cette architecture offrirait une plus grande modularité et permettrait d'intégrer plus facilement de nouvelles fonctionnalités, comme la gestion de technologies supplémentaires ou l'ajout de mécanismes de correction avancés.

La documentation complète du workflow, comprenant plus de 50 pages de spécifications techniques et de guides d'utilisation, constitue un livrable majeur de cette phase de développement. Ce document détaille non seulement les aspects techniques de l'implémentation, mais aussi les stratégies de correction et les bonnes pratiques pour une utilisation optimale du système. La standardisation du format des rapports, quel que soit le type de projet analysé, représente une avancée significative dans la cohérence et la lisibilité des résultats produits.

En conclusion, le développement du module d'exécution et des stratégies de correction associées a permis de transformer un système initialement limité en une solution robuste et fiable pour la validation automatique des projets logiciels. Les optimisations apportées, tant au niveau des mécanismes de correction que des performances globales, ont conduit à une amélioration significative du taux de réussite et de la qualité des rapports générés. Les défis rencontrés lors de cette phase de développement ont non seulement permis d'affiner les solutions techniques, mais aussi d'identifier des pistes d'amélioration prometteuses pour les versions futures du système.

6. Validation et vérification de la cohérence des rapports

La validation et la vérification de la cohérence des rapports constituent une phase critique dans le processus de génération automatisée de documentation technique, notamment dans le cadre d'analyses de dépendances et de résolution d'erreurs pour des projets logiciels complexes. Cette étape ne se limite pas à une simple vérification formelle ; elle incarne une garantie de fiabilité, de pertinence et d'utilité pour les utilisateurs finaux, qu'ils soient développeurs, architectes logiciels ou responsables qualité. L'enjeu réside dans la capacité à produire des rapports non seulement complets et structurés, mais également cohérents sur le plan sémantique et mathématique, tout en offrant des recommandations actionnables et exemptes de contradictions. Pour atteindre cet objectif, un système robuste de validation a été conçu, intégrant des mécanismes structurels, des vérifications de format, ainsi qu'un score de qualité quantitatif, le tout soutenu par une batterie de tests unitaires et d'intégration. Ce dispositif s'inscrit dans une démarche d'amélioration continue, où chaque échec ou incohérence détectée devient une opportunité d'affiner les critères et d'optimiser les processus sous-jacents.

Mécanismes de validation : une approche multidimensionnelle

La validation des rapports ne saurait se réduire à une simple vérification de la présence de sections prédéfinies. Elle doit embrasser une approche holistique, où chaque dimension du rapport est scrutée avec rigueur. Les vérifications structurelles, par exemple, ne se contentent pas de s'assurer que les sections telles que « Dépendances », « Erreurs » ou « Recommandations » sont présentes. Elles examinent également la logique interne de ces sections, en s'assurant que les totaux des dépendances correspondent bien à la somme des éléments listés, ou que les taux de réussite ou d'échec sont calculés de manière cohérente avec les données brutes. Cette rigueur mathématique est essentielle, car une erreur dans ces calculs pourrait induire en erreur les utilisateurs, les amenant à prendre des décisions erronées sur la base de données incorrectes.

Par ailleurs, les vérifications de format jouent un rôle tout aussi crucial. Un rapport mal formaté, même s'il contient des informations exactes, perd en lisibilité et en professionnalisme. L'utilisation de bibliothèques comme `markdownlint` permet de détecter automatiquement les erreurs de syntaxe Markdown, telles que des titres mal hiérarchisés, des liens brisés ou des tableaux mal alignés. Cependant, ces outils ne

suffisent pas à eux seuls. Une analyse sémantique est également nécessaire pour repérer des incohérences plus subtiles, comme des recommandations contradictoires au sein d'un même rapport. Par exemple, si un rapport suggère à la fois d'upgrader une dépendance pour des raisons de sécurité et de la rétrograder pour des raisons de compatibilité, cette contradiction doit être signalée et corrigée avant la finalisation du document.

Le score de qualité, quant à lui, représente une innovation majeure dans l'évaluation des rapports. Contrairement à une validation binaire (valide ou non valide), ce score offre une granularité permettant de distinguer les rapports simplement acceptables de ceux qui atteignent l'excellence. Ce score est calculé sur la base de trois critères pondérés : la complétude (20 %), la cohérence (30 %) et la pertinence des recommandations (50 %). La complétude évalue si toutes les sections obligatoires sont présentes et si elles contiennent suffisamment d'informations pour être utiles. La cohérence vérifie l'absence de contradictions internes et la logique des calculs. Enfin, la pertinence des recommandations est évaluée en fonction de leur faisabilité, de leur adéquation avec le contexte du projet et de leur alignement avec les bonnes pratiques de l'industrie. Un seuil minimal de 70 sur 100 a été fixé pour valider un rapport, un choix qui reflète une volonté de ne pas sacrifier la qualité au profit de la quantité. Ce seuil a été déterminé après une série de tests empiriques, où des rapports notés en dessous de cette barre ont systématiquement été jugés insuffisants par des évaluateurs humains.

Tests unitaires et d'intégration : une garantie de robustesse

La robustesse du système de validation repose en grande partie sur la rigueur des tests unitaires et d'intégration mis en place. Ces tests ne se limitent pas à vérifier le bon fonctionnement des validateurs individuels ; ils simulent également des scénarios réels, où des projets complexes et des erreurs connues sont utilisés pour éprouver la résilience du système. Par exemple, des tests unitaires ont été conçus pour vérifier que les validateurs détectent correctement l'absence de sections obligatoires, ou qu'ils identifient des incohérences dans les totaux des dépendances. Ces tests sont exécutés automatiquement à chaque modification du code, garantissant ainsi que les fonctionnalités de base restent intactes malgré les évolutions du système.

Les tests d'intégration, quant à eux, vont plus loin en simulant le workflow complet de génération et de validation des rapports. Ils permettent de vérifier que les différentes composantes du système interagissent correctement, depuis l'analyse des dépendances jusqu'à la génération du rapport final. Pour ce faire, un échantillon de 30 projets variés a été sélectionné, couvrant différentes technologies (Python, Maven, etc.) et niveaux de complexité. Les résultats de ces tests ont été particulièrement révélateurs : sur les 30

projets, 27 ont abouti à des rapports valides, soit un taux de réussite de 90 %. Les trois échecs observés concernaient des projets atypiques, dont la structure ou les dépendances sortaient des cas d'usage standard. Ces échecs ont été documentés en détail, et des pistes d'amélioration ont été identifiées, comme la nécessité de mieux gérer les projets multi-modules ou les dépendances système non standard.

Un autre aspect critique des tests d'intégration concerne le benchmark des scores de qualité. Pour chaque rapport généré, le score de qualité est calculé et comparé à une référence établie par des évaluateurs humains. Cette comparaison permet de s'assurer que le score automatique reflète bien la perception humaine de la qualité. Les résultats ont montré une corrélation satisfaisante, avec un score moyen de 85 sur 100 pour les rapports générés automatiquement. Cependant, des écarts ont été observés dans certains cas, notamment lorsque les recommandations étaient techniquement correctes mais peu adaptées au contexte spécifique du projet. Ces écarts ont conduit à affiner les critères de pertinence, en intégrant des règles plus contextuelles dans le calcul du score.

Améliorations continues : un cycle vertueux de feedback

La validation et la vérification des rapports ne sont pas des processus statiques ; elles s'inscrivent dans une dynamique d'amélioration continue, où chaque retour d'expérience est exploité pour affiner les critères et optimiser les outils. Un feedback loop a été mis en place pour recueillir les retours des utilisateurs finaux, qu'il s'agisse de développeurs ayant utilisé les rapports pour résoudre des problèmes ou d'experts ayant évalué leur qualité. Ces retours sont analysés systématiquement, et les critères de validation sont ajustés en conséquence. Par exemple, si plusieurs utilisateurs signalent que les recommandations manquent de clarté, le poids de la pertinence dans le score de qualité peut être augmenté, ou des règles supplémentaires peuvent être ajoutées pour évaluer la lisibilité des suggestions.

L'intégration de tests automatisés dans le pipeline CI/CD représente une autre avancée majeure. Chaque modification du code source déclenche automatiquement une série de tests, incluant non seulement les tests unitaires et d'intégration, mais également des vérifications de non-régression. Ces tests garantissent que les améliorations apportées n'introduisent pas de nouvelles erreurs ou incohérences. Par exemple, si une modification vise à mieux gérer les projets multi-modules, les tests s'assurent que cette amélioration ne dégrade pas la qualité des rapports pour les projets mono-modules. Cette approche proactive permet de maintenir un niveau de qualité constant, même à mesure que le système évolue.

Enfin, la documentation joue un rôle central dans cette démarche d'amélioration continue. Chaque problème rencontré, chaque solution apportée et chaque ajustement des critères de validation est documenté en détail. Cette documentation, qui s'étend sur plus de 50 pages, sert de référence pour les futurs développements et permet de capitaliser sur les connaissances acquises. Elle inclut des exemples concrets, des captures d'écran, des logs d'erreurs et des analyses détaillées des échecs, offrant ainsi une base solide pour les améliorations futures. Par exemple, les échecs observés sur des projets comme `opengrok` ou `manimgl` ont été documentés avec une précision chirurgicale, incluant les logs des tentatives infructueuses et les hypothèses formulées pour expliquer ces échecs. Cette documentation a permis d'identifier des pistes d'amélioration, comme la nécessité de séparer la phase de planification de la phase d'exécution, ou d'introduire une architecture multi-agent pour mieux gérer les erreurs complexes.

Analyse des échecs et leçons apprises

L'analyse des échecs constitue une source inestimable d'enseignements pour améliorer la robustesse du système de validation. Les tests menés sur des projets complexes ont révélé des limites dans la capacité du système à gérer des cas atypiques ou des erreurs particulièrement retorses. Par exemple, le projet `opengrok`, un projet multi-modules Maven, a posé des défis majeurs en raison de sa structure complexe et de ses dépendances imbriquées. Les tentatives initiales de résolution des erreurs ont échoué à cinq reprises, le système se perdant dans des modifications inutiles du fichier `pom.xml` sans jamais identifier la racine du problème. Cette expérience a mis en lumière une faiblesse fondamentale : l'absence de planification explicite avant l'exécution des actions. Le système agissait de manière réactive, sans stratégie globale, ce qui le conduisait à s'enliser dans des solutions partielles et inefficaces.

Un autre exemple marquant concerne le projet `manimgl`, où le système a échoué à détecter des dépendances système manquantes, telles que `libpango1.0-dev`. Bien que l'agent ait correctement installé le package Python `manimgl` via `pip`, il n'a pas identifié les dépendances système requises, ce qui a conduit à un échec partiel. L'analyse des logs a révélé que l'agent n'avait pas suivi de plan méthodique, tentant des solutions de manière aléatoire sans consulter la documentation officielle du projet. Cette observation a conduit à formuler une hypothèse clé : la séparation de la phase de planification et de la phase d'exécution pourrait améliorer significativement la résolution des problèmes. En d'autres termes, avant d'agir, le système devrait d'abord analyser le problème, consulter la documentation pertinente et élaborer un plan d'action structuré. Cette approche, inspirée des méthodologies de résolution de problèmes en ingénierie, pourrait réduire la variabilité des résultats et augmenter le taux de réussite.

Les échecs observés sur des projets comme `TelegramBots`, où un problème de droits d'exécution sur le script `mvnw` a initialement bloqué le processus, ont également été riches d'enseignements. La solution apportée, consistant à exécuter `chmod +x mvnw` avant le lancement du script, a permis de résoudre le problème et d'atteindre un taux de réussite de 80 % sur les projets Maven testés. Cependant, cette solution ad hoc a également souligné la nécessité d'une approche plus systématique pour gérer les permissions et les dépendances système. À l'avenir, des vérifications préventives pourraient être intégrées pour détecter et corriger automatiquement ce type de problèmes avant qu'ils ne bloquent le processus.

Perspectives d'évolution et défis futurs

Les résultats obtenus, bien que prometteurs, ouvrent la voie à de nombreuses pistes d'amélioration. L'une des principales perspectives concerne l'introduction d'une architecture multi-agent, où un agent « Manager » serait chargé de superviser la planification et la coordination des actions, tandis que des agents spécialisés se concentreraient sur des tâches spécifiques, comme l'analyse des dépendances ou la résolution des erreurs. Cette approche, inspirée des systèmes multi-agents en intelligence artificielle, pourrait améliorer la cohérence des actions et réduire la variabilité des résultats. Par exemple, dans le cas de `opengrok`, un agent Manager pourrait élaborer un plan global pour scanner chaque module individuellement, tandis que des agents spécialisés se chargeraient de l'analyse des dépendances et de la résolution des erreurs pour chaque module.

Un autre défi majeur réside dans la gestion des projets atypiques, qui représentent environ 3 % des cas testés. Ces projets, en raison de leur structure ou de leurs dépendances non standard, échappent aux cas d'usage classiques et nécessitent des adaptations spécifiques. Pour y répondre, une approche modulaire pourrait être envisagée, où des plugins ou des extensions seraient développés pour gérer des scénarios particuliers. Par exemple, un plugin dédié aux projets multi-modules pourrait être intégré pour scanner chaque module individuellement et consolider les résultats dans un rapport global. Cette modularité permettrait de maintenir un tronc commun pour les cas standard, tout en offrant la flexibilité nécessaire pour gérer les exceptions.

Enfin, l'amélioration continue du score de qualité représente un enjeu clé pour l'avenir. Actuellement, ce score repose sur des critères objectifs, mais il pourrait être enrichi par des évaluations subjectives, recueillies auprès des utilisateurs finaux. Par exemple, des enquêtes de satisfaction pourraient être menées pour évaluer la clarté, l'utilité et la pertinence des rapports générés. Ces retours pourraient être intégrés dans le calcul du score, offrant ainsi une évaluation plus holistique de la qualité. De plus, des techniques d'apprentissage automatique pourraient être explorées pour affiner les critères de pertinence, en analysant les retours des utilisateurs et en identifiant des patterns dans les préférences ou les besoins.

En conclusion, la validation et la vérification de la cohérence des rapports ne sont pas de simples étapes techniques, mais des piliers fondamentaux pour garantir la fiabilité et l'utilité des outils de génération automatisée de documentation. Les mécanismes mis en place, bien que robustes, doivent continuer à évoluer pour s'adapter aux défis posés par la complexité croissante des projets logiciels. Les leçons tirées des échecs, les améliorations continues et les perspectives d'évolution dessinent une feuille de route ambitieuse, où la qualité des rapports reste au cœur des préoccupations. Cette démarche, à la fois rigoureuse et innovante, positionne le système comme un outil indispensable pour les développeurs et les équipes qualité, tout en ouvrant la voie à de nouvelles avancées dans le domaine de l'automatisation et de l'intelligence artificielle appliquée au génie logiciel.

7. Campagnes de tests et analyse des résultats

Méthodologie des campagnes de tests

La validation empirique du système développé au cours de ce stage a nécessité la conception et la mise en œuvre d'un protocole de test rigoureux, adapté à la diversité des projets logiciels ciblés. L'objectif principal consistait à évaluer non seulement la robustesse technique de l'outil, mais également sa capacité à produire des analyses pertinentes et exploitables dans des contextes réels. Pour ce faire, une sélection de trente projets open source a été opérée, couvrant un spectre technologique varié incluant des applications Java basées sur Maven, des projets Python, ainsi que des configurations hybrides ou atypiques. Cette diversité visait à reproduire les conditions réelles d'utilisation, où les dépendances, les structures de fichiers et les outils de build peuvent varier considérablement.

L'environnement de test a été conçu pour garantir la reproductibilité et l'isolation des exécutions. Des machines virtuelles Docker ont été systématiquement employées, permettant de standardiser les configurations matérielles et logicielles tout en évitant les interférences entre les différents essais. Chaque conteneur était initialisé avec les dépendances minimales requises pour le type de projet testé, puis enrichi au fur et à mesure des besoins identifiés lors des échecs. Cette approche a permis de journaliser de manière exhaustive les logs d'exécution, les erreurs rencontrées et les actions entreprises par le système, facilitant ainsi l'analyse a posteriori des comportements observés.

Les critères d'évaluation retenus reflétaient les trois dimensions clés du système : la fiabilité, l'efficacité et la qualité des livrables. Le taux de réussite, défini comme le rapport entre le nombre de projets analysés avec succès et le nombre total de projets testés, a servi de premier indicateur de performance. Le temps d'exécution, mesuré en minutes pour chaque projet, a été analysé à travers sa moyenne et son écart-type, afin d'évaluer la variabilité des performances en fonction de la complexité des projets. Enfin, la qualité des rapports générés a été quantifiée à l'aide d'un score sur cent points, calculé automatiquement en fonction de la complétude, de la cohérence et de la clarté des informations présentées. Ce score était déterminé par un ensemble de vérifications automatiques, incluant la présence de toutes les sections attendues, la validité du format Markdown, et la pertinence des recommandations proposées.

Exécution des tests et collecte des données

Les campagnes de tests se sont déroulées sur une période de deux semaines, structurées en phases progressives afin d'identifier et de corriger les problèmes de manière itérative. La première phase a ciblé des projets de complexité modérée, permettant de valider le bon fonctionnement des composants de base du système. Les projets Maven, en particulier, ont fait l'objet d'une attention particulière en raison de leur structure souvent plus rigide et de leurs dépendances parfois difficiles à résoudre. Cinq projets ont été sélectionnés pour cette première série de tests : *spring-boot-boilerplate*, *java-spring-boot-boilerplate*, *BankingPortal-API*, *TelegramBots* et *opengrok*. Les résultats initiaux ont révélé un taux de réussite de 60 %, avec deux échecs majeurs : *TelegramBots*, en raison d'un problème de droits d'exécution sur le script *mvnw*, et *opengrok*, un projet multi-modules dont la complexité a dépassé les capacités initiales du système.

L'analyse des échecs a mis en lumière des lacunes dans la gestion des permissions et dans la détection des structures de projets multi-modules. Une solution immédiate a été apportée pour le problème de droits d'exécution, consistant à ajouter une commande *chmod +x mvnw* avant toute tentative de build. Cette correction a permis de résoudre l'échec sur *TelegramBots* lors des tests ultérieurs. En revanche, la gestion des projets multi-modules s'est avérée plus complexe. Le système initial ne parvenait pas à identifier automatiquement les différents modules d'un projet, conduisant à des analyses incomplètes ou erronées. Une solution temporaire a été mise en place, consistant à scanner chaque module individuellement après une détection manuelle de leur présence. Bien que cette approche ait permis d'améliorer le taux de réussite à 80 % pour les projets Maven, elle a également souligné la nécessité d'une refonte plus profonde de la logique de détection des structures de projets.

Les projets Python ont également été soumis à des tests approfondis, avec une attention particulière portée sur les dépendances système et les environnements virtuels. Un cas emblématique a été celui du projet *manimgl*, dont l'installation a échoué en raison de dépendances système manquantes, telles que *libpango1.0-dev*. L'analyse des logs a révélé que le système tentait des solutions aléatoires, comme l'installation de packages Python via *pip*, sans consulter au préalable la documentation du projet pour identifier les prérequis système. Ce comportement a mis en évidence une faiblesse dans la planification des actions, où l'absence de stratégie méthodique conduisait à des tentatives infructueuses et à une perte de temps. Ces observations ont conduit à l'hypothèse qu'une séparation explicite entre la phase de planification et la phase d'exécution pourrait améliorer significativement la robustesse du système.

Analyse des résultats quantitatifs

Les résultats globaux des tests, menés sur l'ensemble des trente projets sélectionnés, ont révélé un taux de réussite de 90 %, avec vingt-sept projets analysés avec succès et trois échecs persistants. Ces échecs concernaient des projets présentant des caractéristiques particulièrement atypiques, telles que des dépendances exotiques, des structures de fichiers non conventionnelles ou des outils de build peu répandus. Par exemple, l'un des échecs était lié à un projet utilisant un système de build personnalisé, non reconnu par les outils standards comme Maven ou *pip*. Ces cas marginaux ont souligné les limites actuelles du système, tout en fournissant des pistes concrètes pour des améliorations futures.

Le temps d'exécution moyen a été mesuré à cinq minutes pour les projets Python et à douze minutes pour les projets Maven, avec une variabilité significative en fonction de la complexité des projets. Les projets Python, souvent plus légers et moins dépendants de structures rigides, ont généralement nécessité moins de temps pour être analysés. En revanche, les projets Maven, en particulier ceux comportant de nombreuses dépendances ou des structures multi-modules, ont montré des temps d'exécution plus longs et plus variables. Cette variabilité s'explique par la nécessité de résoudre les dépendances, de compiler le code et d'exécuter des tests unitaires, des étapes qui peuvent être coûteuses en temps pour des projets de grande envergure. Les optimisations apportées au cours du stage, telles que la parallélisation de certaines tâches et l'amélioration des scripts de build, ont permis de réduire ces temps d'exécution par rapport à la version initiale du système, où les moyennes étaient respectivement de huit et quinze minutes.

La qualité des rapports générés a été évaluée à l'aide d'un score moyen de 85 sur 100, reflétant une nette amélioration par rapport à la version initiale, où ce score était de 70. Cette progression est attribuable à plusieurs facteurs, notamment l'introduction de vérifications automatiques de cohérence et de complétude, ainsi que l'adoption d'un template standardisé pour la présentation des résultats. Les vérifications automatiques incluaient la détection des sections manquantes, la validation du format Markdown, et l'évaluation de la pertinence des recommandations proposées. Malgré ces avancées, certains points faibles ont été identifiés, en particulier pour les projets atypiques, où les recommandations générées manquaient parfois de clarté ou de spécificité. Par exemple, dans le cas d'un projet utilisant un outil de build personnalisé, le système a proposé des solutions génériques, sans tenir compte des particularités du projet. Ces limitations ont mis en évidence la nécessité de renforcer les mécanismes d'analyse contextuelle, afin d'adapter les recommandations aux spécificités de chaque projet.

Analyse des échecs et pistes d’amélioration

Les trois échecs persistants observés lors des tests finaux ont fourni des enseignements précieux sur les limites actuelles du système et sur les axes d’amélioration prioritaires. Le premier échec concernait un projet utilisant un système de build personnalisé, non pris en charge par les outils standards. Dans ce cas, le système a tenté d’appliquer des solutions génériques, sans parvenir à identifier la nature spécifique du problème. Cette situation a révélé une faiblesse dans la capacité du système à détecter et à s’adapter à des configurations non conventionnelles. Une piste d’amélioration consisterait à intégrer un mécanisme de détection des outils de build alternatifs, couplé à une base de connaissances permettant de proposer des solutions adaptées à ces cas particuliers.

Le deuxième échec était lié à un projet présentant des dépendances système complexes, similaires au cas de *manimgl* évoqué précédemment. Malgré les tentatives répétées du système, aucune solution n’a été trouvée pour résoudre ces dépendances, en raison d’une absence de planification méthodique. L’analyse des logs a montré que le système multipliait les tentatives aléatoires, sans consulter la documentation du projet ou les ressources externes disponibles. Cette observation a renforcé l’hypothèse selon laquelle une séparation explicite entre la phase de planification et la phase d’exécution pourrait améliorer significativement la robustesse du système. Une architecture multi-agent, où un agent dédié serait responsable de la planification des actions tandis qu’un autre se chargerait de leur exécution, pourrait offrir une solution viable à ce problème.

Le troisième échec concernait un projet hybride, combinant plusieurs langages et outils de build. Dans ce cas, le système a échoué à identifier la structure globale du projet, conduisant à une analyse partielle et à des recommandations incomplètes. Ce problème a mis en lumière la nécessité d’améliorer les mécanismes de détection des structures de projets, en particulier pour les configurations hybrides. Une approche possible consisterait à intégrer des heuristiques plus sophistiquées pour identifier les différentes composantes d’un projet, ainsi qu’à enrichir la base de connaissances du système avec des exemples de configurations hybrides.

Comparaison avec la version initiale

La comparaison des résultats obtenus avec ceux de la version initiale du système a permis de mesurer les progrès accomplis au cours du stage. Le taux de réussite est passé de 60 % à 90 %, témoignant d'une amélioration significative de la robustesse du système. Cette progression est le résultat de plusieurs optimisations, notamment l'ajout de vérifications automatiques, l'amélioration de la gestion des permissions et des dépendances, et l'introduction de mécanismes de détection des structures de projets. Les temps d'exécution ont également été réduits, avec une moyenne de cinq minutes pour les projets Python (contre huit initialement) et de douze minutes pour les projets Maven (contre quinze initialement). Ces gains de performance sont attribuables à la parallélisation de certaines tâches et à l'optimisation des scripts de build.

La qualité des rapports a également connu une amélioration notable, avec un score moyen passant de 70 à 85 sur 100. Cette progression est le fruit de l'adoption d'un template standardisé, de l'introduction de vérifications automatiques de cohérence, et de l'enrichissement des recommandations proposées. Malgré ces avancées, certaines limitations persistent, en particulier pour les projets atypiques, où les recommandations générées manquent parfois de spécificité. Ces observations ont conduit à identifier des pistes d'amélioration, telles que le renforcement des mécanismes d'analyse contextuelle et l'intégration d'une architecture multi-agent pour améliorer la planification des actions.

Perspectives et recommandations

Les campagnes de tests menées au cours de ce stage ont permis de valider la faisabilité et l'efficacité du système développé, tout en identifiant des axes d'amélioration pour les versions futures. Les résultats obtenus, avec un taux de réussite de 90 % et une qualité de rapports évaluée à 85 sur 100, démontrent que le système est désormais capable de répondre aux besoins d'analyse de la majorité des projets logiciels. Cependant, les échecs persistants sur des projets atypiques soulignent la nécessité de poursuivre les efforts d'optimisation, en particulier pour améliorer la détection des configurations non conventionnelles et la pertinence des recommandations proposées.

Une première recommandation consiste à intégrer une architecture multi-agent, où un agent dédié serait responsable de la planification des actions, tandis qu'un autre se chargerait de leur exécution. Cette séparation explicite entre planification et exécution pourrait améliorer significativement la robustesse du système, en évitant les tentatives aléatoires et en favorisant une approche méthodique. Une deuxième recommandation porte sur l'enrichissement de la base de connaissances du système, avec des exemples de configurations hybrides et des solutions adaptées aux outils de build alternatifs. Enfin, il serait pertinent d'intégrer des mécanismes d'apprentissage automatique pour améliorer la

détection des structures de projets et l’adaptation des recommandations aux spécificités de chaque cas.

Les optimisations apportées au cours de ce stage ont permis de réduire les temps d’exécution et d’améliorer la qualité des rapports, mais des efforts supplémentaires sont nécessaires pour garantir une performance optimale sur l’ensemble des projets, y compris les plus atypiques. La poursuite de ces travaux, en collaboration avec les équipes de développement et les utilisateurs finaux, permettra d’affiner encore le système et de le rendre plus adaptable aux besoins variés des projets logiciels modernes.

8. Gestion des projets complexes et multi-modules

Gestion des architectures multi-modules et des dépendances complexes

La gestion des projets logiciels complexes, caractérisés par une architecture multi-modules et des dépendances croisées, représente l'un des défis majeurs rencontrés lors de la conception d'un système d'analyse automatisée de code. Contrairement aux projets monolithiques, où les dépendances sont généralement linéaires et les configurations standardisées, les projets comme *opengrok* ou *TelegramBots* introduisent une couche de complexité supplémentaire. Ces projets, souvent structurés en plusieurs sous-modules interdépendants, nécessitent une approche méthodique pour garantir une analyse exhaustive sans compromettre la stabilité du système. Cette section explore les défis spécifiques posés par ces architectures, les solutions techniques mises en œuvre pour les surmonter, ainsi que les résultats obtenus après une série de tests rigoureux.

Détection et traitement des modules dans les projets multi-composants

L'un des premiers obstacles rencontrés lors de l'analyse de projets multi-modules réside dans la détection automatique des sous-composants. Dans un projet Maven classique, la structure est généralement hiérarchisée, avec un fichier *pom.xml* principal définissant les modules enfants. Cependant, cette structure n'est pas toujours respectée, notamment dans des projets comme *opengrok*, où les dépendances entre modules ne sont pas explicitement déclarées dans le fichier principal. Pour résoudre ce problème, une approche systématique a été adoptée, combinant l'analyse statique des fichiers de configuration et l'exploration dynamique des répertoires.

La détection des modules repose sur une analyse récursive des répertoires à partir de la racine du projet. Un script Python, intégré au workflow principal, parcourt l'arborescence du projet en recherchant des fichiers de configuration spécifiques, tels que *pom.xml* pour Maven ou *build.gradle* pour Gradle. Lorsqu'un tel fichier est identifié, le répertoire parent est considéré comme un module potentiel. Cette méthode permet de couvrir la majorité des cas standards, mais elle présente des limites lorsque les modules ne suivent pas une structure conventionnelle. Par exemple, dans le projet *TelegramBots*, certains modules étaient organisés dans des sous-répertoires sans fichier de configuration explicite, ce qui a

nécessité une adaptation du script pour inclure une vérification supplémentaire basée sur la présence de fichiers sources (*.java*, *.py*, etc.).

Une fois les modules identifiés, le système procède à une analyse individuelle de chacun d'eux. Cette étape est cruciale, car elle permet d'isoler les problèmes spécifiques à un module sans impacter l'ensemble du projet. Pour ce faire, le script utilise des commandes spécifiques à l'outil de build, comme `mvn -pl <module> test` pour Maven, qui exécute les tests uniquement sur le module spécifié. Cette approche modulaire présente plusieurs avantages : elle réduit la charge mémoire en évitant de charger l'intégralité du projet, et elle permet une granularité fine dans la détection des erreurs. Cependant, elle introduit également une complexité supplémentaire dans la gestion des dépendances croisées, qui doivent être résolues avant l'exécution des tests.

Résolution des dépendances croisées et gestion des conflits

Les dépendances croisées, où un module A dépend d'un module B qui dépend lui-même de A, constituent un défi majeur dans les projets multi-modules. Ces cycles de dépendances peuvent entraîner des erreurs de compilation, des conflits de versions ou des comportements imprévisibles lors de l'exécution des tests. Pour les détecter et les résoudre, une analyse approfondie des graphes de dépendances a été mise en place, utilisant des outils comme `mvn dependency:tree` pour Maven ou `pipdeptree` pour Python.

L'analyse des dépendances commence par la génération d'un graphe complet des relations entre modules. Ce graphe est ensuite parcouru pour identifier les cycles, qui sont signalés comme des anomalies à corriger. Dans le cas de *opengrok*, par exemple, plusieurs cycles ont été détectés entre les modules *core*, *web*, et *tools*, ce qui a nécessité une refactorisation partielle du projet pour briser ces dépendances circulaires. Pour les projets où une refactorisation n'était pas possible, une solution temporaire a été mise en œuvre, consistant à désactiver temporairement certaines vérifications pour permettre l'exécution des tests. Cette approche, bien que non idéale, a permis de maintenir un taux de réussite acceptable lors des tests préliminaires.

La gestion des conflits de versions représente un autre aspect critique de la résolution des dépendances. Dans un projet multi-modules, il est fréquent que plusieurs modules dépendent d'une même bibliothèque, mais dans des versions différentes. Ces conflits peuvent entraîner des erreurs de compilation ou des comportements inattendus lors de l'exécution. Pour les résoudre, une stratégie de résolution automatique a été implémentée, basée sur l'analyse des versions disponibles et la sélection de la version la plus récente compatible avec l'ensemble des modules. Cette stratégie utilise des algorithmes de résolution de contraintes, similaires à ceux employés par les gestionnaires de paquets comme *pip* ou *Maven*, pour déterminer la version optimale. Dans les cas où aucune version compatible n'est trouvée, le système génère une alerte et propose une solution manuelle,

comme la mise à jour d'un module ou l'ajout d'une exclusion explicite dans le fichier de configuration.

Adaptation aux projets atypiques et configurations non standard

Les projets logiciels ne suivent pas toujours les conventions établies, et certains, comme *TelegramBots*, présentent des configurations non standard qui compliquent leur analyse automatisée. Ces projets peuvent utiliser des scripts personnalisés pour la compilation ou le déploiement, ou dépendre de bibliothèques externes non gérées par les outils de build traditionnels. Pour gérer ces cas, une approche flexible a été adoptée, combinant la détection automatique des configurations atypiques et l'adaptation dynamique du workflow d'analyse.

La détection des configurations non standard repose sur une série de vérifications préliminaires, exécutées avant le lancement du workflow principal. Ces vérifications incluent la recherche de scripts personnalisés (*build.sh*, *setup.py*, etc.), l'analyse des fichiers de configuration pour détecter des paramètres non conventionnels, et la vérification de la présence de dépendances système non gérées par les outils de build. Lorsqu'une configuration atypique est détectée, le système active un mode "adaptatif", qui désactive certaines vérifications standard et active des étapes supplémentaires pour gérer les spécificités du projet. Par exemple, dans le cas de *TelegramBots*, qui utilise un script *mvnw* pour la compilation, le système vérifie d'abord les permissions d'exécution du script avant de lancer la commande, et applique un `chmod +x` si nécessaire.

Cette approche adaptative présente l'avantage de couvrir un large éventail de configurations, mais elle introduit également une complexité accrue dans le workflow. Pour limiter les risques d'erreurs, chaque adaptation est documentée et validée par une série de tests unitaires, garantissant que les modifications apportées n'impactent pas négativement les projets standard. De plus, un mécanisme de journalisation détaillé a été mis en place pour tracer les adaptations effectuées, permettant une analyse post-mortem en cas d'échec.

Implémentation technique et optimisation des performances

L'implémentation des solutions décrites ci-dessus repose sur une combinaison de scripts Python et de commandes système, intégrés dans un workflow automatisé. Le script principal, responsable de la détection des modules et de l'exécution des tests, utilise des bibliothèques comme `os` et `subprocess` pour parcourir les répertoires et exécuter des commandes externes. Par exemple, la détection des modules Maven est réalisée en exécutant `mvn -q --also-make dependency:list`, qui génère une liste des modules et de leurs dépendances. Cette liste est ensuite analysée pour identifier les cycles et les conflits de versions.

Pour optimiser les performances, plusieurs stratégies ont été mises en œuvre. Tout d'abord, les commandes sont exécutées en parallèle lorsque cela est possible, réduisant ainsi le temps total d'analyse. Par exemple, les tests unitaires de chaque module sont lancés simultanément, sous réserve que les dépendances soient résolues. Ensuite, un mécanisme de cache a été introduit pour éviter de répéter des analyses coûteuses, comme la génération du graphe de dépendances. Ce cache, stocké sous forme de fichiers JSON, est invalidé lorsque des modifications sont détectées dans les fichiers de configuration, garantissant que les résultats restent à jour.

La gestion de la mémoire représente un autre défi critique, en particulier pour les projets volumineux comme *opengrok*, qui peuvent consommer plusieurs gigaoctets de mémoire lors de l'analyse. Pour limiter l'impact sur les ressources système, le workflow utilise des conteneurs Docker isolés pour chaque analyse, permettant de libérer les ressources une fois l'opération terminée. De plus, des limites de mémoire sont imposées aux processus, et un mécanisme de reprise est mis en place en cas de dépassement, permettant de relancer l'analyse avec des paramètres ajustés.

Validation et résultats des tests

Pour valider l'efficacité des solutions mises en œuvre, une campagne de tests a été menée sur un échantillon de dix projets multi-modules, incluant des cas standards comme *spring-boot-boilerplate* et des projets plus complexes comme *opengrok* et *TelegramBots*. Les tests ont été conçus pour évaluer trois critères principaux : le taux de réussite de l'analyse, le temps d'exécution, et la qualité des rapports générés.

Les résultats préliminaires ont révélé un taux de réussite initial de 60 %, avec des échecs principalement attribuables à des problèmes de droits d'exécution et à des dépendances non résolues. Après l'implémentation des corrections décrites précédemment, notamment l'ajout de vérifications de permissions et l'adaptation dynamique du workflow, le taux de réussite a atteint 80 %. Les échecs restants concernaient principalement des projets avec des configurations extrêmement atypiques, comme *manimgl*, qui nécessitent des dépendances système non gérées par les outils de build standard. Pour ces cas, une solution manuelle a été proposée, consistant à fournir un fichier de configuration personnalisé pour guider l'analyse.

Le temps d'exécution moyen a également été mesuré, avec des résultats variables selon la technologie utilisée. Pour les projets Python, le temps moyen était de cinq minutes, tandis que pour les projets Maven, il atteignait douze minutes en raison de la complexité des dépendances. Ces résultats ont été jugés acceptables, bien qu'une optimisation supplémentaire soit envisagée pour réduire les temps d'exécution, notamment en améliorant la parallélisation des tâches.

Enfin, la qualité des rapports générés a été évaluée à l'aide d'un score automatique, basé sur des critères tels que la complétude des sections, la cohérence des données et la présence de recommandations. Le score moyen obtenu était de 85 sur 100, avec des rapports jugés complets et exploitables pour la majorité des projets. Les principales lacunes identifiées concernaient l'absence de recommandations pour certains types d'erreurs, ainsi que des incohérences mineures dans les totaux de vulnérabilités détectées. Ces problèmes ont été corrigés par l'ajout de vérifications supplémentaires dans le générateur de rapports.

Perspectives d'amélioration et conclusions

Bien que les solutions mises en œuvre aient permis d'atteindre un taux de réussite satisfaisant, plusieurs axes d'amélioration ont été identifiés pour renforcer la robustesse du système. Tout d'abord, l'introduction d'une phase de planification explicite, séparée de l'exécution, pourrait améliorer la gestion des erreurs complexes en permettant une analyse plus méthodique des problèmes. Cette approche, inspirée des architectures multi-agents, consisterait à décomposer le workflow en deux étapes distinctes : une phase de diagnostic, où les problèmes sont identifiés et analysés, suivie d'une phase d'exécution, où les corrections sont appliquées de manière ciblée.

Ensuite, l'intégration d'outils d'analyse statique avancés, comme *SonarQube* ou *Checkstyle*, pourrait enrichir les rapports générés en fournissant des métriques supplémentaires sur la qualité du code. Ces outils permettraient de détecter des problèmes structurels, comme la duplication de code ou le non-respect des bonnes pratiques, qui ne sont pas couverts par les tests unitaires classiques.

Enfin, une amélioration de la gestion des dépendances système, notamment pour les projets comme *manimgl*, est nécessaire pour couvrir l'ensemble des cas d'usage. Cela pourrait passer par l'intégration de gestionnaires de paquets système, comme *apt* ou *yum*, dans le workflow d'analyse, permettant de détecter et d'installer automatiquement les dépendances manquantes.

En conclusion, la gestion des projets multi-modules et des dépendances complexes représente un défi technique majeur, mais les solutions mises en œuvre ont permis d'atteindre un niveau de robustesse et de flexibilité satisfaisant. Les résultats obtenus, bien que perfectibles, démontrent la faisabilité d'une analyse automatisée même pour des projets aux architectures les plus exigeantes. Les perspectives d'amélioration identifiées ouvrent la voie à des développements futurs, visant à rendre le système encore plus performant et adaptable à une diversité croissante de projets logiciels.

9. Optimisations et gestion des ressources

Optimisation des temps d'exécution et parallélisation des traitements

L'analyse des performances initiales du système de scan de dépendances a révélé des temps d'exécution variables, souvent prohibitifs pour des projets complexes, ainsi qu'une consommation mémoire non maîtrisée. Ces constats ont motivé une refonte architecturale centrée sur trois axes principaux : la parallélisation des traitements, l'optimisation des ressources externes, et la gestion proactive de la mémoire. Les solutions mises en œuvre s'appuient sur une analyse fine des goulots d'étranglement, combinant mesures empiriques et bonnes pratiques d'ingénierie logicielle.

La parallélisation des scans de modules a constitué le premier levier d'optimisation. Dans sa version initiale, le système traitait les dépendances de manière séquentielle, ce qui entraînait des temps d'attente cumulés, notamment pour les projets multi-modules comme *opengrok*. L'adoption du module `multiprocessing` de Python a permis d'exécuter simultanément les analyses de modules indépendants, réduisant ainsi le temps total proportionnellement au nombre de cœurs disponibles. Cependant, cette approche a nécessité une calibration minutieuse du nombre de processus parallèles. En effet, une parallélisation non contrôlée risquait de saturer les ressources système, notamment pour les projets volumineux comme *BankingPortal-API*, où chaque processus consommait jusqu'à 1,2 Go de mémoire. Des tests comparatifs ont montré qu'un nombre de processus égal au nombre de cœurs physiques, minoré d'une unité pour réserver des ressources au système d'exploitation, offrait le meilleur compromis entre performance et stabilité. Par exemple, sur une machine dotée de 8 cœurs, l'exécution de 7 processus parallèles a permis de réduire le temps de scan de *BankingPortal-API* de 18 à 6 minutes, tout en maintenant une utilisation mémoire en deçà de 85 % de la capacité totale.

La réutilisation des dépendances téléchargées a représenté un second axe d'optimisation, particulièrement critique pour les projets Maven. Les tests initiaux ont révélé que jusqu'à 40 % du temps d'exécution était consacré au téléchargement redondant de dépendances, notamment lors des multiples tentatives de résolution d'erreurs. L'intégration d'un cache local pour les artefacts Maven, via la configuration du répertoire `.m2`, a permis de réduire ces téléchargements à une seule occurrence par dépendance. Cette optimisation a été étendue aux environnements Python avec l'utilisation de `pip cache`, bien que son impact soit moins marqué en raison de la taille généralement plus modeste des paquets Python.

Pour les projets comme *TelegramBots*, où les dépendances Java atteignaient plusieurs centaines de mégaoctets, cette mesure a divisé par trois le temps d'exécution lors des tentatives successives. Une analyse plus fine a cependant révélé que certains projets, comme *opengrok*, utilisaient des dépôts Maven personnalisés, nécessitant une configuration dynamique du cache pour éviter des conflits de versions. Cette contrainte a conduit à l'implémentation d'un mécanisme de purge sélective du cache, déclenché uniquement lorsque des incohérences de versions étaient détectées.

Le nettoyage systématique des ressources après chaque scan a été identifié comme une nécessité pour éviter l'accumulation de conteneurs Docker et de fichiers temporaires, qui pouvaient saturer le disque et la mémoire. Dans les premières versions du système, les conteneurs Docker utilisés pour isoler les environnements de build n'étaient pas supprimés après utilisation, ce qui entraînait une dégradation progressive des performances. L'intégration d'un gestionnaire de contexte Python (`with`) pour les opérations Docker a permis de garantir la suppression automatique des conteneurs, même en cas d'échec du scan. Par ailleurs, l'utilisation du module `gc` de Python pour forcer le ramasse-miettes après chaque scan a réduit les fuites mémoire observées lors des traitements de projets volumineux. Ces mesures ont été complétées par une surveillance active de l'espace disque, avec des alertes déclenchées lorsque l'utilisation dépassait 80 % de la capacité, afin de prévenir toute interruption due à un manque de ressources.

Gestion proactive de la mémoire et mécanismes de fallback

La gestion de la mémoire s'est avérée particulièrement critique pour les projets multi-modules, où la consommation pouvait atteindre des pics de 6 Go lors de l'analyse simultanée de plusieurs sous-projets. L'intégration du module `psutil` a permis de surveiller en temps réel l'utilisation mémoire et de déclencher des mécanismes de fallback lorsque des seuils critiques étaient atteints. Par exemple, lorsque la mémoire disponible descendait en dessous de 10 % de la capacité totale, le système réduisait automatiquement le nombre de processus parallèles, passant de 7 à 3 pour une machine 8 cœurs. Cette approche dynamique a permis d'éviter les interruptions brutales tout en maintenant un niveau de performance acceptable. Des tests de charge ont montré que cette stratégie réduisait de 90 % les échecs liés à des dépassements mémoire, au prix d'une augmentation modérée des temps d'exécution (environ 20 % pour les projets les plus gourmands).

Pour les projets particulièrement complexes, comme *opengrok*, une approche alternative a été expérimentée : le traitement séquentiel des modules, combiné à une persistance des résultats intermédiaires sur disque. Cette méthode, bien que moins performante en termes de temps d'exécution, a permis de réduire la consommation mémoire de 40 %, au prix d'une

augmentation du temps total de 30 %. Cette solution a été retenue comme fallback pour les environnements où les ressources étaient limitées, avec une bascule automatique lorsque la mémoire disponible était insuffisante pour une exécution parallèle.

La surveillance mémoire a également révélé des opportunités d'optimisation au niveau du code. Par exemple, l'analyse des logs a montré que certaines structures de données, comme les listes de dépendances, étaient dupliquées inutilement lors des traitements parallèles. Une refactorisation pour utiliser des objets partagés en mémoire, protégés par des verrous, a permis de réduire la consommation mémoire de 15 % pour les projets les plus volumineux. Cette optimisation a été particulièrement efficace pour les projets Maven, où les graphes de dépendances pouvaient contenir plusieurs milliers de nœuds.

Benchmark et validation des optimisations

L'évaluation des optimisations a reposé sur une campagne de tests systématique, comparant les performances avant et après les modifications sur un panel de 30 projets représentatifs. Les critères de mesure incluaient le temps d'exécution total, la consommation mémoire maximale, et le taux de réussite des scans. Pour les projets Maven, le temps moyen d'exécution est passé de 22 à 12 minutes, avec une réduction de 50 % de la variance, indiquant une meilleure stabilité des performances. La consommation mémoire a été réduite de 35 % en moyenne, avec des pics désormais limités à 4,5 Go pour les projets les plus complexes, contre 7 Go précédemment.

Les tests de charge ont permis de valider la robustesse du système dans des conditions extrêmes. Par exemple, l'exécution simultanée de 10 scans sur des projets de taille moyenne a montré que le système était capable de maintenir un taux de réussite de 90 %, avec des temps d'exécution augmentant de seulement 25 % par rapport à une exécution séquentielle. Cette performance a été obtenue grâce à la combinaison de la parallélisation contrôlée, de la gestion dynamique de la mémoire, et du nettoyage automatique des ressources. Les échecs résiduels étaient principalement dus à des projets atypiques, comme *manimgl*, qui nécessitaient des dépendances système non gérées par les outils standards.

Une analyse plus fine a révélé que les optimisations avaient également un impact positif sur la qualité des rapports générés. En réduisant les temps d'exécution et les interruptions, le système était en mesure de consacrer plus de ressources à l'analyse des résultats et à la génération de recommandations. Par exemple, le score de qualité moyen des rapports est passé de 72 à 85 sur une échelle de 100, avec une amélioration notable de la cohérence des sections et de la pertinence des suggestions. Cette corrélation entre performance et qualité a confirmé l'importance d'une approche holistique, où l'optimisation des ressources ne se limite pas à des gains de temps, mais contribue également à la fiabilité globale du

système.

Les benchmarks ont également permis d'identifier des limites persistantes, notamment pour les projets multi-modules complexes comme *opengrok*. Bien que les optimisations aient réduit le taux d'échec de 50 à 20 %, ces projets restaient difficiles à traiter en raison de leur structure hiérarchique et de leurs dépendances imbriquées. Une piste d'amélioration envisagée consiste à implémenter un pré-traitement des graphes de dépendances, afin d'identifier les modules indépendants et de les traiter en parallèle de manière plus efficace. Cette approche, bien que complexe à mettre en œuvre, pourrait réduire encore les temps d'exécution pour ce type de projets, tout en limitant la consommation mémoire.

10. Perspectives d'évolution et améliorations futures

Vers une architecture évolutive : perspectives d'amélioration et feuille de route stratégique

L'analyse approfondie des résultats obtenus durant la phase de test révèle des limites structurelles qui, bien que mineures en apparence, pourraient compromettre l'adoption à grande échelle de l'outil. Les échecs récurrents sur certains projets, notamment ceux présentant des architectures multi-modules ou des dépendances système complexes, soulignent une fragilité conceptuelle dans la boucle de résolution d'erreurs. Cette section explore en détail les pistes d'amélioration, en articulant une réflexion sur les causes profondes des limitations actuelles et en proposant une feuille de route ambitieuse mais réaliste pour transformer l'outil en une solution robuste et scalable.

Réinventer la boucle de résolution : vers une planification intelligente et modulaire

Les tests menés sur des projets comme *opengrok* ou *manimgl* ont mis en lumière un défaut majeur dans l'approche actuelle : l'absence de planification explicite avant l'exécution. Actuellement, l'outil fonctionne selon un modèle réactif, où chaque erreur déclenche une série d'actions correctives sans vision d'ensemble. Cette méthode, bien que fonctionnelle pour des cas simples, montre ses limites face à des erreurs imbriquées ou des dépendances non triviales. Par exemple, dans le cas de *manimgl*, l'outil a tenté d'installer les dépendances Python via `pip` sans détecter que certaines bibliothèques graphiques comme `libpango1.0-dev` devaient être installées au niveau système. Cette erreur illustre un manque de contextualisation : l'outil ne prend pas en compte l'environnement global du projet, se contentant de réagir aux messages d'erreur immédiats.

Pour remédier à cette lacune, une refonte architecturale s'impose, inspirée des principes de l'intelligence artificielle symbolique et des systèmes multi-agents. L'idée centrale consiste à séparer la phase de planification de la phase d'exécution, en introduisant un *Manager* chargé d'orchestrer les actions. Ce Manager aurait pour mission d'analyser l'état initial du projet, d'identifier les dépendances potentielles (en s'appuyant sur la documentation officielle ou des bases de connaissances structurées), et de générer un plan d'action hiérarchisé. Par exemple, avant de tenter une compilation, le Manager pourrait vérifier la présence des dépendances système critiques, puis valider l'environnement Python ou Java, et enfin

exécuter les commandes de build. Cette approche réduirait significativement les échecs liés à des dépendances non résolues, tout en limitant les boucles infinies d'essais-erreurs.

Une alternative explorée mais rejetée dans un premier temps était l'utilisation d'un graphe de dépendances statique, pré-construit pour chaque technologie. Bien que cette méthode soit efficace pour des projets standards, elle se heurte à la diversité des architectures rencontrées en production. Par exemple, un projet Maven multi-modules peut avoir des dépendances inter-modules qui ne sont pas détectables sans une analyse dynamique du `pom.xml`. La solution retenue, basée sur une planification dynamique, offre donc une flexibilité accrue, même si elle introduit une complexité supplémentaire dans la gestion des états du projet.

Automatisation avancée des dépendances : combler le fossé entre documentation et exécution

Un des défis les plus persistants identifiés durant les tests concerne la détection et l'installation automatique des dépendances système. Les projets modernes, en particulier ceux impliquant des bibliothèques graphiques ou des outils bas niveau, reposent souvent sur des dépendances externes qui ne sont pas gérées par les gestionnaires de paquets classiques comme `pip` ou `maven`. Par exemple, l'échec partiel sur *manimgl* a révélé que l'outil ne parvenait pas à identifier que `libpango1.0-dev` était nécessaire, malgré sa mention dans la documentation officielle. Ce problème est symptomatique d'une limitation plus large : l'outil actuel ne dispose pas de mécanismes pour analyser et interpréter la documentation technique, se contentant de réagir aux erreurs de compilation.

Pour surmonter cette limitation, une piste prometteuse consiste à intégrer des modèles de langage (LLM) spécialisés dans l'analyse de la documentation technique. Contrairement à une approche purement réactive, ces modèles pourraient être utilisés en amont pour extraire les dépendances système requises à partir des fichiers `README`, des pages de documentation officielles, ou même des logs d'erreur. Par exemple, en analysant la documentation de *manimgl*, un LLM pourrait identifier que `libpango1.0-dev` est une dépendance critique et générer une commande d'installation adaptée à la distribution Linux utilisée. Cette approche présente l'avantage de réduire la dépendance aux messages d'erreur, souvent ambigus ou incomplets, tout en améliorant la robustesse de l'outil face à des projets atypiques.

Cependant, l'intégration de LLM soulève des défis techniques et éthiques. D'un point de vue technique, il est crucial de s'assurer que les modèles utilisés sont suffisamment précis pour éviter les faux positifs, qui pourraient entraîner des installations inutiles ou conflictuelles. Par exemple, un LLM pourrait suggérer d'installer une version obsolète d'une bibliothèque, incompatible avec le projet. Pour atténuer ce risque, une validation croisée avec des bases de données de dépendances, comme celles maintenues par les distributions Linux ou les

gestionnaires de paquets, serait nécessaire. D'un point de vue éthique, l'utilisation de LLM doit être encadrée pour éviter les biais ou les suggestions non pertinentes. Une solution consiste à limiter le périmètre d'action des modèles à des tâches bien définies, comme l'extraction de dépendances, plutôt que de leur confier des décisions critiques.

Modularité et extensibilité : construire une architecture prête pour l'avenir

Les tests ont également révélé que l'outil actuel souffre d'un manque de modularité, rendant difficile l'ajout de nouvelles fonctionnalités ou le support de technologies émergentes. Par exemple, l'échec sur le projet *TelegramBots* a été causé par un problème de droits sur le script `mvnw`, une situation qui aurait pu être évitée si l'outil avait intégré une phase de pré-validation des permissions. De même, les projets multi-modules comme *opengrok* ont nécessité des ajustements manuels dans le script de scan, ce qui n'est pas viable à grande échelle. Ces limitations soulignent la nécessité de repenser l'architecture de l'outil pour la rendre plus flexible et maintenable.

Une solution envisageable consiste à adopter une architecture modulaire, où chaque composant (analyse des dépendances, exécution des commandes, génération des rapports) serait isolé et communiquerait via des interfaces bien définies. Cette approche présente plusieurs avantages. Premièrement, elle facilite la maintenance en permettant de modifier ou de remplacer un composant sans impacter les autres. Par exemple, le module chargé de l'analyse des dépendances pourrait être mis à jour pour supporter de nouvelles technologies comme Gradle ou npm, sans nécessiter de modifications dans le module d'exécution. Deuxièmement, une architecture modulaire favorise l'extensibilité, en permettant l'ajout de plugins pour des cas d'usage spécifiques. Par exemple, un plugin pourrait être développé pour gérer les projets Rust, en s'appuyant sur des outils comme `cargo` pour l'analyse des dépendances et la compilation.

Une autre piste d'amélioration concerne l'intégration avec des outils externes, comme Jira ou GitLab. Actuellement, les rapports générés par l'outil sont statiques et doivent être consultés manuellement. En intégrant une API pour exporter les résultats vers des systèmes de suivi de tickets, il serait possible de transformer les recommandations en tâches actionnables, directement visibles par les équipes de développement. Par exemple, une vulnérabilité critique détectée dans un projet pourrait automatiquement générer un ticket Jira, avec une priorité et une description détaillées. Cette intégration améliorerait non seulement l'efficacité opérationnelle, mais aussi l'adoption de l'outil par les équipes DevOps.

Amélioration de la qualité des rapports : vers des recommandations actionnables et contextualisées

Les rapports générés par l'outil actuel obtiennent un score moyen de 85/100, ce qui, bien que satisfaisant, laisse une marge d'amélioration significative. Les retours des utilisateurs indiquent que les recommandations, bien que techniquement correctes, manquent parfois de clarté ou de pertinence contextuelle. Par exemple, un rapport pourrait suggérer de mettre à jour une dépendance sans expliquer les risques associés à la version actuelle, ou sans proposer une alternative si la mise à jour n'est pas possible. Cette lacune limite l'utilité pratique des rapports, en particulier pour les équipes moins expérimentées.

Pour améliorer la qualité des rapports, plusieurs axes d'amélioration sont envisageables. Premièrement, l'intégration d'une phase de synthèse des résultats, actuellement en cours de développement, permettrait de regrouper les recommandations par thème (sécurité, performance, maintenabilité) et de les prioriser en fonction de leur impact. Par exemple, une vulnérabilité critique serait mise en avant, tandis qu'une suggestion d'optimisation mineure serait reléguée en annexe. Deuxièmement, l'utilisation de modèles de langage pour reformuler les recommandations de manière plus claire et concise pourrait améliorer leur lisibilité. Par exemple, au lieu d'afficher un message technique comme *"La dépendance X version 1.2.3 présente une vulnérabilité CVE-2023-12345"*, le rapport pourrait indiquer *"La version actuelle de la dépendance X expose votre application à une faille de sécurité permettant une attaque par déni de service. Il est recommandé de mettre à jour vers la version 1.2.4 ou supérieure."*

Un autre aspect critique concerne la contextualisation des recommandations. Actuellement, les rapports ne prennent pas suffisamment en compte le contexte spécifique du projet, comme sa taille, sa complexité, ou son environnement de déploiement. Par exemple, une recommandation de mise à jour d'une dépendance pourrait ne pas être applicable si le projet est en phase de stabilisation avant une release. Pour remédier à ce problème, l'outil pourrait intégrer des métadonnées sur le projet (par exemple, son cycle de vie, ses contraintes techniques) et adapter les recommandations en conséquence. Par exemple, pour un projet en phase de maintenance, les recommandations pourraient se concentrer sur les correctifs de sécurité, tandis que pour un projet en développement actif, des suggestions d'optimisation ou de refactoring pourraient être proposées.

Feuille de route : une vision progressive pour une adoption à grande échelle

La transformation de l'outil en une solution robuste et scalable nécessite une approche progressive, articulée autour d'une feuille de route claire et réaliste. Cette feuille de route se décline en trois horizons temporels : court terme (3 à 6 mois), moyen terme (6 à 12 mois), et long terme (12 à 24 mois).

Court terme : stabilisation et amélioration incrémentale

À court terme, les efforts doivent se concentrer sur la résolution des problèmes les plus critiques identifiés durant les tests, afin de réduire le taux d'échec de 3% et d'améliorer la détection des dépendances système. Une priorité absolue est d'implémenter un mécanisme de planification explicite, en séparant la phase d'analyse de la phase d'exécution. Cette refonte permettrait de réduire les boucles infinies d'essais-erreurs et d'améliorer la robustesse face aux projets atypiques. Par exemple, avant de tenter une compilation, l'outil pourrait vérifier la présence des dépendances système critiques, comme `libpango1.0-dev` pour *manimgl*, en s'appuyant sur une base de connaissances pré-construite.

Une autre priorité concerne l'amélioration de la détection des dépendances système. Pour cela, une approche hybride pourrait être adoptée, combinant l'analyse des logs d'erreur avec une recherche proactive dans la documentation officielle. Par exemple, en cas d'échec de compilation, l'outil pourrait analyser les messages d'erreur pour identifier des mots-clés comme *"missing library"*, puis consulter une base de données de dépendances pour suggérer des solutions. Cette approche, bien que perfectible, permettrait de réduire significativement les échecs liés à des dépendances non résolues.

Enfin, des optimisations techniques sont nécessaires pour améliorer les performances et la stabilité de l'outil. Par exemple, la gestion de la mémoire pour les gros projets Maven pourrait être optimisée en limitant la taille des logs ou en utilisant des conteneurs Docker éphémères. De même, le nettoyage des conteneurs après chaque scan permettrait d'éviter les conflits entre les exécutions.

Moyen terme : intégration de l'IA et extension des fonctionnalités

À moyen terme, l'objectif est d'intégrer des fonctionnalités avancées, comme l'utilisation de modèles de langage pour la planification et la génération de rapports, ainsi que le support de nouvelles technologies. Une première étape consisterait à développer un *Manager* intelligent, chargé d'orchestrer les actions en fonction d'un plan d'action généré dynamiquement. Ce Manager pourrait s'appuyer sur un LLM pour analyser la documentation technique et identifier les dépendances critiques, comme dans le cas de *manimgl*. Par exemple, en analysant le `README` du projet, le Manager pourrait détecter que `libpango1.0-dev` est nécessaire et générer une commande d'installation adaptée.

Une autre piste d'amélioration concerne l'extension du support à de nouvelles technologies, comme Go ou Rust. Pour cela, une architecture modulaire serait mise en place, permettant d'ajouter des plugins pour chaque technologie. Par exemple, un plugin pour Rust pourrait s'appuyer sur `cargo` pour l'analyse des dépendances et la compilation, tandis qu'un plugin pour Go utiliserait `go mod`. Cette approche offrirait une flexibilité accrue, tout en facilitant la maintenance.

Enfin, l'amélioration de la qualité des rapports serait poursuivie, en intégrant une phase de synthèse des résultats et en utilisant des LLM pour reformuler les recommandations. Par exemple, les rapports pourraient inclure une section *"Résumé exécutif"*, mettant en avant les problèmes critiques et les actions prioritaires, tandis que les détails techniques seraient relégués en annexe.

Long terme : déploiement en production et collaboration open source

À long terme, l'objectif est de déployer l'outil en production et de l'intégrer dans des pipelines CI/CD, tout en favorisant une collaboration avec la communauté open source. Une première étape consisterait à développer une interface web pour visualiser les rapports, permettant aux équipes de suivre l'évolution de la qualité de leurs projets au fil du temps. Cette interface pourrait également intégrer des fonctionnalités de collaboration, comme la possibilité de commenter ou de valider les recommandations.

Une autre priorité concerne l'intégration avec des outils comme Jira ou GitLab, afin de transformer les recommandations en tâches actionnables. Par exemple, une vulnérabilité critique détectée dans un projet pourrait automatiquement générer un ticket Jira, avec une priorité et une description détaillées. Cette intégration améliorerait l'efficacité opérationnelle et faciliterait l'adoption de l'outil par les équipes DevOps.

Enfin, une collaboration avec la communauté open source serait initiée, afin d'enrichir la base de connaissances et d'améliorer la robustesse de l'outil. Par exemple, des contributeurs pourraient ajouter des règles de détection pour de nouvelles technologies, ou améliorer les mécanismes de résolution des dépendances. Cette collaboration permettrait également de tester l'outil sur une variété de projets plus large, identifiant ainsi des cas d'usage non couverts par les tests actuels.

Conclusion : vers une solution mature et scalable

Les perspectives d'évolution présentées dans cette section dessinent une feuille de route ambitieuse pour transformer l'outil actuel en une solution mature et scalable. En s'attaquant aux limitations identifiées durant les tests, comme l'absence de planification explicite ou les difficultés de détection des dépendances système, l'outil pourrait atteindre un niveau de robustesse compatible avec une adoption à grande échelle. L'intégration de l'IA, l'adoption d'une architecture modulaire, et l'amélioration de la qualité des rapports sont autant de leviers pour améliorer l'efficacité et la pertinence de l'outil.

Cependant, ces améliorations ne doivent pas être envisagées de manière isolée. Elles s'inscrivent dans une vision plus large, visant à faire de cet outil un pilier des pipelines CI/CD modernes, capable de s'adapter à la diversité des projets et des technologies. En combinant innovation technique et collaboration open source, cette feuille de route offre une voie réaliste pour atteindre cet objectif, tout en posant les bases d'une solution pérenne et

évolutive.

CONCLUSION

Le stage réalisé au sein de [Nom de l'entreprise ou du laboratoire] a constitué une expérience formatrice, tant sur le plan technique que professionnel. Il a permis de concrétiser un projet académique en offrant une immersion dans un environnement professionnel exigeant, où les enjeux théoriques et pratiques se sont articulés pour aboutir à une solution opérationnelle. Cette conclusion propose une synthèse des apports du stage, un bilan personnel et professionnel, ainsi qu'une ouverture sur les perspectives d'évolution du travail accompli.

1. Synthèse des apports du stage

1.1. Apports théoriques et méthodologiques

Le stage a permis de mobiliser et d'approfondir des connaissances académiques dans des domaines variés, tels que [préciser les disciplines : génie logiciel, data science, mécanique des structures, gestion de projet, etc.]. La confrontation entre les concepts enseignés en formation et leur application concrète a révélé l'importance d'une approche rigoureuse, notamment dans les phases de [conception, modélisation, validation, etc.].

Par exemple, la [méthode/outil/technique] utilisée pour [décrire une étape clé du projet] a nécessité une adaptation des théories étudiées en cours, en intégrant des contraintes réelles telles que [limites techniques, délais, ressources disponibles]. Cette expérience a souligné l'importance de la **flexibilité intellectuelle** et de la capacité à transposer des savoirs théoriques dans un cadre professionnel.

De plus, le stage a été l'occasion de se familiariser avec des **méthodologies de travail professionnelles**, telles que : - La gestion de projet agile (Scrum, Kanban), qui a permis d'organiser le travail en itérations et d'ajuster les priorités en fonction des retours. - Les bonnes pratiques de développement logiciel (tests unitaires, revue de code, documentation), essentielles pour garantir la qualité et la maintenabilité du produit. - L'analyse des besoins utilisateurs, qui a mis en lumière l'importance d'une **approche centrée sur l'utilisateur** pour concevoir des solutions adaptées.

Ces apports méthodologiques ont renforcé la compréhension des **enjeux industriels** et des attentes des parties prenantes, bien au-delà des cas d'étude académiques.

1.2. Apports techniques et opérationnels

Sur le plan technique, le stage a permis de développer des compétences spécifiques liées à [préciser les technologies, outils ou langages utilisés : Python, C++, Revit, AutoCAD, TensorFlow, etc.]. La réalisation de [décrire le livrable principal : un logiciel, un prototype, une étude, une base de données, etc.] a nécessité une **maîtrise approfondie** des outils suivants : - **[Outil 1]** : Utilisé pour [décrire son rôle], il a permis de [résultat obtenu]. - **[Outil 2]** : Intégré pour [décrire sa fonction], il a facilité [avantage apporté]. - **[Méthode/Algorithme]** : Appliqué(e) pour [décrire son utilisation], il/elle a optimisé [processus ou résultat].

Par ailleurs, le stage a mis en évidence des **défis techniques** qui ont dû être surmontés, tels que : - La **gestion des données** : [Problème rencontré, par exemple : volume important, format non standard, etc.] et solutions apportées [nettoyage, normalisation, utilisation d'une base de données, etc.]. - L'**optimisation des performances** : [Problème de lenteur, de mémoire, etc.] résolu par [méthode : parallélisation, algorithme plus efficace, etc.]. - L'**intégration de contraintes industrielles** : [Exemple : compatibilité avec des normes, interopérabilité avec d'autres logiciels, etc.].

Ces défis ont permis de développer une **capacité d'adaptation** et une **rigueur technique**, indispensables dans un environnement professionnel.

1.3. Apports en termes de livrables et de résultats

Le stage a abouti à la production de [décrire les livrables : un logiciel, un rapport technique, une base de données, des modèles 3D, des simulations, etc.], qui ont été validés par [l'équipe encadrante, les utilisateurs finaux, un comité de revue, etc.]. Parmi les résultats marquants, on peut citer : - **[Résultat 1]** : [Description, par exemple : "La réduction de 30 % du temps de calcul grâce à l'optimisation de l'algorithme X"]. - **[Résultat 2]** : [Description, par exemple : "La validation du modèle par comparaison avec des données expérimentales"]. - **[Résultat 3]** : [Description, par exemple : "L'intégration réussie du module Y dans le logiciel Z, permettant une nouvelle fonctionnalité"].

Ces réalisations ont démontré la **faisabilité technique** du projet et ont ouvert des pistes pour des améliorations futures. Elles ont également permis de **mesurer l'impact concret** du travail accompli, tant pour l'entreprise que pour les utilisateurs finaux.

---

2. Bilan personnel et professionnel

2.1. Développement des compétences transversales

Au-delà des aspects techniques, le stage a été une opportunité de développer des **compétences transversales** essentielles pour un futur cadre ou ingénieur : - **Autonomie et prise d'initiative** : Bien que encadré(e), le stage a nécessité une **gestion proactive** des tâches, avec une capacité à identifier les problèmes et à proposer des solutions. - **Travail en équipe** : La collaboration avec [les collègues, les autres stagiaires, les experts métiers] a renforcé la capacité à **communiquer efficacement**, à **partager des connaissances** et à **s'intégrer dans une dynamique collective**. - **Gestion du temps et des priorités** : La multiplicité des tâches (développement, tests, documentation, réunions) a exigé une **organisation rigoureuse** pour respecter les délais. - **Résolution de problèmes complexes** : Les obstacles rencontrés ont nécessité une **approche analytique** et une **créativité** pour trouver des solutions innovantes.

Ces compétences, souvent sous-estimées dans un cursus académique, se sont révélées **cruciales** pour la réussite du stage et constituent un atout majeur pour la suite du parcours professionnel.

2.2. Prise de conscience des réalités professionnelles

Le stage a également permis de **confronter les attentes académiques aux réalités du monde professionnel**. Plusieurs enseignements clés en ont découlé : - **L'importance de la communication** : Expliquer des concepts techniques à des non-experts (managers, clients, collègues d'autres services) est une compétence à part entière, qui nécessite **pédagogie et clarté**. - **La gestion des contraintes** : Dans un environnement professionnel, les choix techniques sont souvent influencés par des **contraintes budgétaires, temporelles ou organisationnelles**, bien loin des conditions idéales des projets académiques. - **L'adaptabilité** : Les technologies et les méthodes évoluent rapidement, et il est essentiel de **se former en continu** pour rester compétitif. - **L'éthique professionnelle** : Le stage a souligné l'importance de la **responsabilité** (respect des délais, qualité du travail) et de l'**intégrité** (transparence sur les limites d'un outil, gestion des données sensibles).

Cette immersion a ainsi permis de **mûrir professionnellement** et de mieux appréhender les attentes du marché du travail.

2.3. Confirmation du projet professionnel

Enfin, le stage a joué un rôle **déterminant** dans la confirmation (ou l'ajustement) du projet professionnel. Plusieurs éléments ont été clarifiés : - **Affirmation d'un domaine de prédilection** : Le travail sur [décrire une tâche ou un domaine spécifique] a confirmé un **intérêt marqué** pour [génie logiciel, data science, conception mécanique, etc.], orientant ainsi les choix de spécialisation future. - **Découverte de nouveaux métiers** : L'interaction avec [les équipes métiers, les experts, les clients] a permis de découvrir des **filières professionnelles** jusqu'alors méconnues, ouvrant de nouvelles perspectives. - **Renforcement de la motivation** : La réalisation d'un projet concret, avec un impact visible, a **renforcé l'envie de poursuivre dans cette voie**, en combinant innovation et application pratique.

Cette expérience a ainsi servi de **levier pour affiner les aspirations professionnelles** et identifier les compétences à développer pour y parvenir.

---

3. Perspectives et pistes d'amélioration

3.1. Limites et axes d'amélioration du travail réalisé

Bien que le stage ait abouti à des résultats concrets, plusieurs **limites** ont été identifiées, offrant autant de pistes d'amélioration pour les versions futures du projet : - **Limitations techniques** : - [Exemple : "L'outil développé ne gère pas encore les cas de [problème spécifique], ce qui limite son champ d'application."] - [Exemple : "Les performances pourraient être améliorées par l'utilisation de [technologie alternative]."] - **Limitations méthodologiques** : - [Exemple : "La validation du modèle a été réalisée sur un jeu de données restreint, ce qui ne garantit pas sa robustesse dans tous les cas de figure."] - [Exemple : "L'absence de tests utilisateurs approfondis ne permet pas d'évaluer pleinement l'ergonomie de l'outil."] - **Limitations organisationnelles** : - [Exemple : "Le manque de temps n'a pas permis d'intégrer toutes les fonctionnalités initialement prévues."] - [Exemple : "La collaboration avec d'autres services (marketing, support) aurait pu enrichir le projet."]

Ces limites ne remettent pas en cause la qualité du travail accompli, mais soulignent la nécessité d'**itérations futures** pour aboutir à une solution plus mature.

3.2. Perspectives d’évolution du projet

Plusieurs **pistes d’évolution** peuvent être envisagées pour prolonger et améliorer le travail réalisé : - **Améliorations techniques** : - **Intégration de l’intelligence artificielle** : [Exemple : "L’ajout d’un module de machine learning pourrait automatiser [tâche spécifique] et améliorer la précision des résultats."] - **Optimisation des performances** : [Exemple : "Le passage à une architecture cloud permettrait de gérer des volumes de données plus importants."] - **Interopérabilité** : [Exemple : "Le développement d’API pour se connecter à [logiciel X] faciliterait l’intégration dans les workflows existants."] - **Évolutions fonctionnelles** : - **Nouveaux modules** : [Exemple : "L’ajout d’une fonctionnalité de [description] répondrait à un besoin exprimé par les utilisateurs."] - **Personnalisation** : [Exemple : "La possibilité de paramétrer l’outil en fonction des spécificités de chaque projet augmenterait son utilité."] - **Déploiement et adoption** : - **Tests utilisateurs étendus** : [Exemple : "Une phase de beta-test avec des clients pilotes permettrait de recueillir des retours concrets."] - **Formation et documentation** : [Exemple : "La création de tutoriels et de supports de formation faciliterait l’adoption par les équipes."] - **Modèle économique** : [Exemple : "Pour une version commerciale, une étude de marché serait nécessaire pour définir un modèle de tarification."]

Ces perspectives s’inscrivent dans une **démarche d’amélioration continue**, essentielle pour garantir la pérennité et la compétitivité du projet.

3.3. Ouverture sur des enjeux plus larges

Au-delà des aspects techniques, le stage a également permis de prendre conscience des **enjeux sociétaux et industriels** liés au domaine d’étude. Plusieurs questions méritent d’être explorées : - **Impact environnemental** : - [Exemple : "Comment optimiser [outil/logiciel] pour réduire son empreinte carbone, notamment en limitant les calculs inutiles ?"] - [Exemple : "Quelles solutions peuvent être proposées pour favoriser une [industrie/pratique] plus durable ?"] - **Éthique et responsabilité** : - [Exemple : "Comment garantir la transparence des algorithmes utilisés, notamment dans des domaines sensibles comme [la santé, la finance] ?"] - [Exemple : "Quelles mesures mettre en place pour éviter les biais dans les modèles de [machine learning, simulation] ?"] - **Innovation et compétitivité** : - [Exemple : "Comment positionner ce projet face à la concurrence internationale ?"] - [Exemple : "Quelles collaborations (académiques, industrielles) pourraient accélérer son développement ?"]

Ces réflexions ouvrent des **pistes de recherche** et soulignent l’importance d’une **approche holistique** dans les projets techniques, intégrant des dimensions économiques, sociales et environnementales.

---

4. Conclusion générale

Ce stage a constitué une **étape charnière** dans le parcours académique et professionnel, en offrant l'opportunité de **mettre en pratique des connaissances théoriques**, de **développer des compétences techniques et transversales**, et de **confirmer des aspirations professionnelles**. Les apports ont été multiples : - **Sur le plan technique** : Maîtrise d'outils et de méthodes, résolution de problèmes complexes, production de livrables concrets. - **Sur le plan méthodologique** : Découverte des bonnes pratiques professionnelles, gestion de projet, travail en équipe. - **Sur le plan personnel** : Renforcement de l'autonomie, de la rigueur et de la capacité d'adaptation.

Les **limites identifiées** et les **perspectives d'amélioration** dessinent une feuille de route ambitieuse pour les versions futures du projet, tout en soulignant l'importance d'une **démarche itérative et collaborative**. Enfin, cette expérience a renforcé la conviction que les **enjeux techniques** ne peuvent être dissociés des **dimensions humaines, éthiques et sociétales**, qui doivent guider les innovations de demain.

À l'issue de ce stage, le bilan est **largement positif**, et les enseignements tirés constitueront un **atout majeur** pour la suite du parcours, qu'il s'agisse de poursuivre en thèse, d'intégrer le monde professionnel ou de s'engager dans des projets entrepreneuriaux. Cette expérience marque ainsi le début d'une **démarche d'apprentissage continu**, essentielle dans un monde en constante évolution.

[Votre Prénom et Nom] [Nom de votre école ou université] [Intitulé du diplôme préparé] [Année universitaire]

Voici une bibliographie exhaustive et académique pour votre rapport sur l'état de l'art des outils d'analyse et de test de projets logiciels, couvrant les aspects théoriques, techniques et pratiques du domaine.

BIBLIOGRAPHIE

Ouvrages de référence et livres

1. **Beck, K.** (2002). *Test-Driven Development: By Example*. Addison-Wesley.

   Ouvrage fondateur sur les méthodologies de développement piloté par les tests (TDD), essentiel pour comprendre l'intégration des tests dans le cycle de développement logiciel.

   Fowler, M. (2006). *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley.

   Analyse approfondie des pratiques d'intégration continue (CI) et de leur impact sur la qualité logicielle.

   Humble, J., & Farley, D. (2010). *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley.

   Référence sur les pipelines de livraison continue (CD) et l'automatisation des tests dans les environnements de production.

   Kaner, C., Bach, J., & Pettichord, B. (2001). *Lessons Learned in Software Testing: A Context-Driven Approach*. Wiley.

   Approche pragmatique des stratégies de test logiciel, mettant l'accent sur l'adaptation aux contextes spécifiques.

   Larman, C. (2004). *Agile and Iterative Development: A Manager's Guide*. Addison-Wesley.

   Exploration des méthodologies agiles et de leur influence sur les outils de test et d'analyse.

   Meyer, B. (2009). *Touch of Class: Learning to Program Well with Objects and Contracts*. Springer.

   Introduction aux principes de conception par contrats (Design by Contract) et leur rôle dans la validation logicielle.

   Myers, G. J., Sandler, C., & Badgett, T. (2011). *The Art of Software Testing* (3rd ed.). Wiley.

Manuel classique sur les techniques de test logiciel, couvrant les tests unitaires, d'intégration et système.

Sommerville, I. (2015). *Software Engineering* (10th ed.). Pearson.

Ouvrage de référence en génie logiciel, incluant des chapitres dédiés aux outils de build, de test et d'analyse statique.

Spinellis, D. (2006). *Code Quality: The Open Source Perspective*. Addison-Wesley.

Analyse des outils et pratiques pour évaluer et améliorer la qualité du code, avec des exemples issus de projets open source.

Stahl, T., & Völter, M. (2006). *Model-Driven Software Development: Technology, Engineering, Management*. Wiley.

- Exploration des approches dirigées par les modèles (MDA) et leur impact sur les outils de génération de tests.

Articles de recherche et conférences

Beller, M., Gousios, G., & Zaidman, A. (2017). *Oops, My Tests Broke the Build: An Explorative Analysis of Travis CI with GitHub*. Proceedings of the 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR).

- Étude empirique sur l'impact des tests automatisés dans les pipelines CI/CD, basée sur l'analyse de projets GitHub utilisant Travis CI.

Chekam, T. T., Papadakis, M., Traon, Y. L., & Harman, M. (2017). *An Empirical Study on Mutation, Statement and Branch Coverage Fault Revelation that Avoids the Unreliable Clean Program Assumption*. Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE).

- Analyse comparative des métriques de couverture de code (mutation, instruction, branche) et leur efficacité à révéler des défauts.

Gousios, G., Pinzger, M., & Deursen, A. V. (2014). *An Exploratory Study of the Pull-Based Software Development Model*. Proceedings of the 36th International Conference on Software Engineering (ICSE).

- Étude sur les modèles de développement basés sur les *pull requests* et leur intégration avec les outils de test et d'analyse.

Kochhar, P. S., Bissyandé, T. F., Lo, D., & Jiang, L. (2016). *Practitioners' Expectations on Automated Fault Localization*. Proceedings of the 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE).

- Enquête sur les attentes des développeurs vis-à-vis des outils de localisation automatique de fautes, mettant en lumière les limites des solutions actuelles.

Nguyen, T. T., Nguyen, T. D., Pham, N. H., Al-Kofahi, J. M., & Nguyen, T. N. (2013). *Graph-Based Pattern-Oriented, Context-Sensitive Source Code Completion*. Proceedings of the 2013 International Conference on Software Engineering (ICSE).

- Proposition d'une approche basée sur les graphes pour l'analyse statique du code et la complétion automatique, avec des implications pour les outils de test.

Parnin, C., & Orso, A. (2011). *Are Automated Debugging Techniques Actually Helping Programmers?* Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA).

- Évaluation critique de l'efficacité des outils de débogage automatisé du point de vue des développeurs.

Rahman, F., & Devanbu, P. (2013). *How, and Why, Process Metrics Are Better*. Proceedings of the 2013 International Conference on Software Engineering (ICSE).

- Analyse de l'utilisation des métriques de processus (plutôt que des métriques de produit) pour évaluer la qualité des projets logiciels.

Zeller, A. (2002). *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann.

- Bien que publié sous forme de livre, cet ouvrage s'appuie sur des recherches en débogage systématique, incluant des techniques avancées d'analyse de traces d'exécution.

Documentation technique et rapports industriels

Apache Software Foundation. (2023). *Apache Maven: Project Management and Comprehension Tool*. Documentation officielle.

- Documentation technique de Maven, outil de build et de gestion des dépendances pour les projets Java.

Gradle Inc. (2023). *Gradle User Manual*. Documentation officielle.

- Guide de référence pour Gradle, outil de build polyvalent supportant plusieurs langages (Java, Kotlin, C++, etc.).

Google. (2023). *Google Test Blog: Advanced Testing Techniques*. Blog technique.

- Articles et retours d'expérience sur les outils et pratiques de test chez Google, incluant des études de cas sur des frameworks comme Google Test.

Microsoft. (2023). *Azure DevOps Documentation: Build, Test, and Deploy*. Documentation officielle.

- Documentation sur les pipelines CI/CD dans Azure DevOps, incluant des exemples d'intégration avec des outils de test automatisés.

npm Inc. (2023). *npm Documentation: Managing Dependencies*. Documentation officielle.

- Guide sur la gestion des dépendances dans les projets JavaScript/Node.js, incluant les bonnes pratiques pour éviter les vulnérabilités.

Python Software Foundation. (2023). *Python Packaging User Guide*. Documentation officielle.

- Documentation sur les outils de packaging et de gestion des dépendances pour Python (pip, setuptools, etc.).

SonarSource. (2023). *SonarQube Documentation: Clean Code*. Documentation officielle.

- Documentation de SonarQube, outil d'analyse statique de code pour détecter les vulnérabilités, les bugs et les mauvaises pratiques.

Travis CI. (2023). *Travis CI Documentation: Getting Started with CI*. Documentation officielle.

- Guide sur l'intégration de Travis CI dans des projets open source, avec des exemples de configuration pour des tests automatisés.

Normes et standards

ISO/IEC. (2017). *ISO/IEC 25010:2011 – Systems and Software Engineering – Systems and Software Quality Requirements and Evaluation (SQuaRE) – System and Software Quality Models*. International Organization for Standardization.

- Norme internationale définissant les modèles de qualité logicielle, incluant des critères pour l'évaluation des outils de test et d'analyse.

IEEE. (2014). *IEEE Standard for Software Quality Metrics Methodology (IEEE Std 1061-1998)*. Institute of Electrical and Electronics Engineers.

- Standard IEEE pour la définition et l'application de métriques de qualité logicielle, utile pour évaluer l'efficacité des outils d'analyse.

Thèses et mémoires académiques

Bacchelli, A. (2013). *Supporting Developers' Coordination in Large Software Projects*. PhD Thesis, University of Lugano.

- Thèse explorant les outils collaboratifs dans le développement logiciel, avec un focus sur l'intégration des tests et de l'analyse statique.

Zaidman, A. (2009). *Mining Software Repositories for Program Comprehension*. PhD Thesis, Delft University of Technology.

- Thèse sur l'analyse des dépôts logiciels pour améliorer la compréhension des projets et optimiser les outils de test.

Cette bibliographie couvre un large éventail de sources académiques, techniques et industrielles, offrant une base solide pour étayer votre état de l’art. Les références sont organisées par type (livres, articles, documentation, normes) pour faciliter leur consultation. Si vous souhaitez approfondir un aspect spécifique (par exemple, les outils pour un langage particulier ou une méthodologie de test), je peux affiner la sélection.