UNIVERSITÉ DU QUÉBEC

MÉMOIRE PRÉSENTÉ À
L'UNIVERSITÉ DU QUÉBEC À TROIS-RIVIÈRES

COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN MATHÉMATIQUES ET INFORMATIQUE APPLIQUÉES

PAR
AMINATA DIOP

GÉNÉRATION DE TESTS UNITAIRES DANS LES SYSTÈMES ORIENTÉS OBJET À L'AIDE DES GRANDS MODÈLES DE LANGAGE

Mai 2025

Résumé

Les tests unitaires sont cruciaux pour garantir la qualité du logiciel, et l'utilisation des grands modèles de langage (LLM) pour automatiser ce processus, peut considérablement améliorer la productivité et la fiabilité du logiciel. Dans le cadre de cette étude, plusieurs approches de génération de tests unitaires ont été comparées, y compris des techniques basées sur le fine-tuning et le prompting de grands modèles de langage (LLM). L'objectif principal était d'évaluer les performances de ces approches à travers des métriques telles que la couverture de lignes, la couverture de branches, et le score de mutation. Les résultats obtenus ont ensuite été comparés à ceux d'EvoSuite, un outil bien établi dans la génération automatique de tests.

L'approche de fine-tuning a obtenu des performances modestes, avec une couverture de lignes de 30,9 %, une couverture de branches de 18,3 % et un score de mutation de 31,2 %. Ces résultats décevants peuvent s'expliquer par la qualité insuffisante des données d'entraînement, notamment un manque de contexte complet et des lacunes dans l'initialisation des objets. Malgré une conformité syntaxique, les tests générés manquaient de pertinence pratique pour détecter des bogues réalistes.

Pour les approches de prompting, deux stratégies ont été testées : une approche globale où la classe (comme unité d'un programme orienté objet) complète est fournie au modèle, et une approche plus ciblée, appelée "contexte focal", où seule la méthode à tester et les signatures des autres méthodes sont incluses. Il a été constaté que l'approche de contexte focal offrait de meilleurs résultats en zero-shot prompting, tandis qu'en few-shot prompting, le contexte global a légèrement surpassé le contexte focal. Cependant, le contexte focal a permis de tester un plus grand nombre de classes, favorisant ainsi une exploration plus large du code.

Par ailleurs, bien que EvoSuite ait surpassé toutes les approches étudiées en termes de couverture de lignes et de branches, l'approche zero-shot prompting, consistant d'abord à lister les scenarios de test pertinents puis à générer les cas de tests unitaires correspondants, a obtenu un score de mutation supérieur, indiquant une meilleure capacité à cibler des mutations (bogues simulés) malgré une couverture globale plus faible. De plus, les tests générés par cette approche se sont avérés plus compréhensibles, soulignant un avantage qualitatif par rapport à EvoSuite.

Abstract

Unit tests are crucial for ensuring software quality, and leveraging large language models (LLM) to automate this process can significantly enhance developer productivity and software reliability. In this study, several unit test generation approaches were compared, including techniques based on fine-tuning and prompting LLM. The primary objective was to evaluate the performance of these approaches using metrics such as line coverage, branch coverage, and mutation score. The results were then compared to those of EvoSuite, a well-established tool for automated test generation.

The fine-tuning approach achieved modest performance, with 30.9% lines of code coverage, 18.2% branch coverage, and a mutation score of 31.2%. These disappointing results may be attributed to the inadequate quality of training data, including a lack of complete context and gaps in object initialization. Although the generated tests adhered to syntactic rules, they lacked practical relevance for detecting realistic bugs (for example testing the factorial of increasing values without any limit).

For prompting-based approaches, two strategies were tested: a global approach, where the complete class (as an object-oriented programming unit) was provided to the model, and a more targeted approach, referred to as "focused context," where only the method to be tested and the signatures of other methods were included. It was observed that the focused context approach delivered better results in zero-shot prompting, whereas in few-shot prompting, the global context slightly outperformed the focused context. However, the focused context allowed testing a larger number of classes, enabling a broader exploration of the code.

Furthermore, although EvoSuite outperformed all the studied approaches in terms of line and branch coverage, the zero-shot prompting approach—first listing relevant test scenarios and then generating the corresponding unit test cases—achieved a higher mutation score, indicating a superior ability to detect mutations (simulated bugs) despite lower overall coverage. Additionally, the tests generated through this approach were found to be more comprehensible, highlighting a qualitative advantage over EvoSuite.

Remerciements

Je tiens à exprimer ma profonde gratitude à toutes les personnes qui ont contribué, de près ou de loin, à la réalisation de ce mémoire.

En premier lieu, je remercie chaleureusement mes directeurs de recherche, Mourad BADRI et Fadel TOURE, pour leur précieuse guidance, leur soutien constant et leurs conseils avisés tout au long de ce travail. Votre expertise et votre disponibilité ont été une source inestimable d'inspiration.

Je souhaite adresser un remerciement spécial à ma famille, qui a toujours été à mes côtés tout au long de ce parcours. Votre amour inconditionnel, vos encouragements constants et votre soutien moral m'ont permis de rester motivée et confiante face aux défis rencontrés.

Un grand merci également à mes amis, particulièrement à Oumoul Vadly AIDARA et à Demba DIOP pour leur présence, leur bienveillance et leur capacité à me remonter le moral lorsque j'en avais besoin. Vous avez contribué à rendre cette aventure plus agréable et supportable.

À tous, merci de m'avoir permis d'atteindre cet objectif avec sérénité et détermination.

Table des matières

RESUME ..... **II**

ABSTRACT ..... **III**

REMERCIEMENTS ..... **IV**

TABLE DES MATIERES ..... **V**

LISTE DES TABLEAUX ..... **VII**

LISTE DES FIGURES ..... **VIII**

LISTE DES ABREVIATIONS ET DES SIGLES ..... **IX**

CHAPITRE 1 ..... **1**

INTRODUCTION **1**

1.1 INTRODUCTION 1

1.2 PROBLÉMATIQUE 2

1.3 ENJEUX 2

1.4 OBJECTIFS DE RECHERCHE 3

1.5 ORGANISATION DU MEMOIRE 4

CHAPITRE 2 ..... **5**

REVUE DE LITTERATURE **5**

2.1 GENERATION GUIDEE PAR LA RECHERCHE (SEARCH-BASED SOFTWARE TESTING, SBST) 5

2.1.1 Travaux clés *5*

2.2 GENERATION ALEATOIRE (RANDOM TESTING) 6

2.2.1 Travaux clés *6*

2.3 OUTILS ALTERNATIFS A EVOSUITE ET RANDOOP 7

2.4 GENERATION GUIDEE PAR L'EXECUTION SYMBOLIQUE (SYMBOLIC EXECUTION) 7

2.5 GENERATION GUIDEE PAR MODELES OU CONTRATS (MODEL-BASED / CONTRACT-BASED) 7

2.6 AVANCEES AVEC LES GRANDS MODELES DE LANGAGE 8

CHAPITRE 3 ..... **10**

APPROCHES EXPLOREES ET CADRE METHODOLOGIQUE **10**

3.1 OUTILS D'EXPERIMENTATION 10

3.1.1 Mistral *10*

3.1.2 JUGE *13*

3.2 APPROCHES EXPLOREES 16

3.2.1 Approche exploratoire initiale : Utilisation des Arbres Syntaxiques Abstraits (AST) *17*

3.2.2 Fine-tuning *20*
3.2.3 Few-Shot Prompting *26*
3.2.4 Zero-Shot prompting *31*
3.3 METHODOLOGIE ADOPTEE : ZERO-SHOT PROMPTING EN DEUX TEMPS (2TZSP) 36
3.4 QUESTIONS DE RECHERCHE 38
3.4.1 Q1 Peut-on générer des tests unitaires pour les benchmarks de JUGE avec les différentes approches ? *38*
3.4.2 Q2 Comment ces approches se comparent-elles à EvoSuite ? *38*
3.4.3 Q3 Quelle est la qualité et la pertinence de ces tests unitaires ? *38*
3.4.4 Q4 Quel est l'impact de la spécification des prompts ? *39*
3.4.5 Q5 Quel est l'impact du contexte spécifique ? *39*

CHAPITRE 4 ..... **40**

ÉVALUATION DES MODELES ET DISCUSSION DES RESULTATS **40**

4.1 RQ1 PEUT-ON GENERER DES TESTS UNITAIRES POUR LES BENCHMARKS DE JUGE AVEC LES DIFFERENTES APPROCHES ? 41
4.2 RQ2 COMMENT CES APPROCHES SE COMPARENT-ELLES À EVOSUITE ? 43
4.2.1 Comparaison entre Zero-Shot prompting en Deux Temps vs Evosuite : *45*
4.2.2 Comparaison entre Few-Shot prompting Contexte Focal vs Evosuite *47*
4.2.3 Comparaison entre Fine-tuning vs Evosuite *49*
4.2.4 Discussion des résultats *50*
4.3 RQ3 QUELLE EST LA QUALITE ET LA PERTINENCE DE CES TESTS UNITAIRES ? 54
4.4 RQ4 QUEL EST L'IMPACT DE LA SPECIFICATION DES PROMPTS ? 56
4.4.1 Amélioration de la qualité des tests *56*
4.4.2 Réduction du bruit et des hallucinations *56*
4.5 RQ5 QUEL EST L'IMPACT DU CONTEXTE FOCAL ? 57

CHAPITRE 5 ..... **59**

CONCLUSION ET PERSPECTIVES **59**

BIBLIOGRAPHIE **61**

Liste des tableaux

Tableau 1 : Ensemble de données Methods2Test 22
Tableau 2 : Statistiques descriptives des projets de SBST 2020 [43] 40
Tableau 3 : performances des approches 42
Tableau 4: "Zero-Shot prompting en Deux Temps" et Evosuite 45
Tableau 5: "Few-shot prompting Contexte Focal" et Evosuite 47
Tableau 6: Fine-tuning et Evosuite 49
Tableau 7: Contexte focal vs Contexte Global (classe complète) 57

Liste des figures

Figure 1 : Performances de Mistral 7B [20] 11
Figure 2 : L'attention par fenêtre coulissante de Mistral 7B [20] 11
Figure 3 : Architecture de JUGE [23] 15
Figure 4 : code source Java de la méthode add 17
Figure 5 : Représentation de l'AST pour la méthode add 18
Figure 6 : L'ensemble Method2test 22
Figure 7 : Le contexte focal ou spécifique 23
Figure 8 : Réorganisation de Method2test 24
Figure 9 : Fine-tuning avec l'API Mistral 25
Figure 10 : prompt du few-shot contexte global 28
Figure 11 : prompt du few-shot contexte focal 30
Figure 12 : prompt du Zero-Shot Contexte Global 33
Figure 13 : prompt Zero-Shot Contexte Focal 35
Figure 14 : phase 1 Zero-Shot prompting en Deux Temps 36
Figure 15 : phase 2 Zero-Shot prompting en Deux Temps 37
Figure 16 : Répartition des tests générés par approche 42
Figure 17 : comparaison entre le 2TZSP et Evosuite 46
Figure 18 : comparaison entre le "Few-shot prompting Contexte Focal" et Evosuite 48
Figure 19 : comparaison entre l'approche Fine-tuning et Evosuite 49

Liste des abréviations et des sigles

LLM: Large Language Model

SBST: Search-Based Software Testing

BERT: Bidirectional Encoder Representations from Transformers

T5: Text-To-Text Transfer Transformer

RANDOOP: Random Tester for Object-Oriented Programs

BART: Bidirectional and Auto-Regressive Transformers

UtGen : Unit Test Generation

2TZSP: Zero-Shot prompting en Deux Temps

Chapitre 1

Introduction

1.1 Introduction

Un test unitaire est une méthode utilisée dans le développement de logiciels pour vérifier individuellement les plus petites parties testables d'une application, appelées unités ou composants. Le but principal d'un test unitaire est de s'assurer que chaque unité de code fonctionne correctement comme prévu. Une "unité" dans le contexte du test unitaire peut varier selon le langage de programmation et le style de conception, mais il s'agit généralement de fonctions ou de méthodes individuelles dans les langages orientés procédure ou objet. Dans les architectures plus complexes, cela peut également inclure des classes entières ou des composants spécifiques.

La génération de tests unitaires constitue une phase cruciale du cycle de développement logiciel, essentielle pour garantir la fiabilité et la qualité des systèmes logiciels. Dans le contexte du développement orienté objet, où les interactions complexes entre objets peuvent introduire des bogues subtiles et parfois critiques, l'importance de tests unitaires efficaces devient encore plus prononcée.

Cependant, la création manuelle de ces tests est souvent une tâche fastidieuse et chronophage, susceptible d'introduire des erreurs humaines et de laisser des parties du code non vérifiées. Face à ces défis, l'automatisation de la génération des tests unitaires émerge comme une solution prometteuse, visant à améliorer l'efficacité du processus de test, à réduire son coût tout en augmentant la couverture et la qualité du code.

L'avènement des LLM, ou "Large Language Models" (grands modèles de langage), comme GPT-4 [1] (Generative Pre-trained Transformer version 4), BERT [2] (Bidirectional Encoder Representations from Transformers), ou encore T5 [3] (Text-To-Text Transfer Transformer), a ouvert de nouvelles perspectives dans l'automatisation de tâches complexes, y compris la génération de tests unitaires. Les LLM sont des modèles d'apprentissage profond, une branche de l'intelligence artificielle qui utilise des réseaux neuronaux profonds pour analyser et apprendre à partir de grandes quantités de données. Ces modèles, basés sur des architectures

comme les Transformers [4], présentent un potentiel prometteur pour produire des tests unitaires adaptés et efficaces, grâce à leur capacité à comprendre et à générer du texte.

1.2 Problématique

Malgré l'importance des tests unitaires, leur création demeure un défi pour de nombreuses équipes de développement. Les outils existants, comme EvoSuite, bien qu'efficaces dans certaines situations, présentent des limites notamment en matière de lisibilité et de compréhension des tests générés.

Dans ce contexte, une question essentielle se pose :

Comment exploiter les grands modèles de langage (LLM) pour générer des tests unitaires qui soient non seulement syntaxiquement corrects et pertinents, mais aussi capables d'améliorer la couverture du code et la détection d'erreurs, tout en surpassant les outils existants ?

Les défis techniques soulevés par cette problématique incluent :

L'identification automatique de cas de test pertinents à partir de référentiels existants.

La génération d'assertions précises et adaptées au contexte.

La production de tests unitaires syntaxiquement corrects, lisibles et capables d'améliorer la couverture du code.

Ainsi, l'utilisation des grands modèles de langage pour la génération automatique de tests unitaires soulève plusieurs défis techniques et méthodologiques. Il est donc essentiel d'examiner les enjeux liés à cette approche afin de mieux comprendre son impact et ses implications.

1.3 Enjeux

Après avoir défini la problématique, cette section vise à identifier les principaux enjeux liés à l'utilisation des grands modèles de langage (LLM) pour la génération automatique de tests unitaires. Ces enjeux peuvent être regroupés en plusieurs catégories :

- **Enjeu technique :**

L'utilisation des LLM pour la génération de tests unitaires doit surmonter plusieurs limitations techniques afin d'être véritablement efficace et applicable à grande échelle.

Qualité et pertinence des tests générés : Démontrer la capacité des LLM à produire des tests de haute qualité, adaptés à des contextes complexes comme le développement orienté objet. Même si un test est syntaxiquement correct, cela ne signifie

pas qu'il est utile. Il doit tester des cas d'usage réels et pertinents, contenir des assertions précises et bien ciblées et ne pas être redondant par rapport aux tests existants.

Optimisation du coût computationnel : Les LLM nécessitent des ressources importantes, et générer des tests de manière efficace implique de trouver le bon équilibre entre qualité et temps de génération.

- **Enjeu méthodologique :**

Comparer différentes approches de génération, notamment le fine-tuning et le prompting, afin de déterminer la plus efficace.

- **Enjeu économique :**

Réduire les coûts et le temps associés à la création manuelle des tests unitaires.

- **Enjeu pratique :**

Proposer des tests unitaires lisibles, pertinents et directement exploitables par les développeurs.

L'utilisation des LLM pour la génération automatique de tests unitaires constitue une avancée prometteuse, mais elle s'accompagne de défis importants. Sur le plan scientifique, il est nécessaire d'améliorer la capacité des modèles à analyser et exploiter la structure du code, ainsi que d'évaluer rigoureusement la qualité des tests générés. Sur le plan technique, il faut veiller à ce que les tests soient non seulement pertinents, mais également lisibles et facilement intégrables dans un projet existant. Sur le plan industriel, il importe que les tests produits répondent aux pratiques et aux standards couramment utilisés par les équipes de développement, afin de faciliter leur utilisation effective. Afin d'adresser ces enjeux, ce travail évalue l'efficacité de différentes approches de génération de tests unitaires à l'aide de LLM, en mesurant la pertinence, la lisibilité et la qualité des tests générés sur la base de métriques reconnues dans la littérature.

1.4 Objectifs de recherche

Ce projet vise à atteindre plusieurs objectifs :

- **Objectif général**

Explorer et évaluer le potentiel des LLM pour automatiser la génération de tests unitaires dans des projets Java. Le langage Java a été choisi pour cette étude de cas en raison de sa large adoption dans l'industrie et dans le milieu académique, ainsi que de la disponibilité de nombreux projets open source de référence. De plus, Java dispose d'un écosystème mature d'outils et de bibliothèques pour la génération et l'évaluation de tests unitaires, ce qui facilite la comparaison avec des outils existants comme EvoSuite. Enfin, un système efficace avec Java

peut être adapté à d'autres langages, car les principes de génération et d'évaluation de tests unitaires restent similaires.

- **Objectifs spécifiques**
    1. Comparer nos deux principales approches : le fine-tuning et le prompting
    2. Évaluer les performances des tests générés à l'aide de métriques standards, notamment : la couverture de lignes et de branches, le score de mutation et la lisibilité et pertinence des tests.
    3. Identifier les forces et faiblesses des LLM par rapport à des outils comme EvoSuite.

Ainsi, ces objectifs guideront notre étude et structureront les différentes analyses menées tout au long de ce travail. Afin de mieux appréhender la démarche adoptée, la section suivante présente l'organisation du mémoire.

1.5 Organisation du mémoire

Afin de répondre à la problématique et aux objectifs définis précédemment, ce mémoire est structuré en plusieurs chapitres, chacun traitant un aspect spécifique du travail réalisé.

Ce premier chapitre met en évidence l'importance de la génération automatique de tests unitaires et introduit le contexte de notre étude. Il définit la problématique de recherche, expose les enjeux liés à cette thématique et précise les objectifs poursuivis.

Le deuxième chapitre fait une revue de la littérature, en présentant les travaux existants sur la génération automatique de tests unitaires. Il expose les approches développées dans ce domaine, en mettant en évidence leurs contributions.

Le troisième chapitre décrit en détail les approches explorées, notamment l'approche Zero-Shot en Deux Temps, ainsi que le protocole d'expérimentation mis en place pour évaluer son efficacité.

Dans le quatrième chapitre, nous présentons les résultats obtenus en comparant notre approche aux outils existants, ainsi qu'une analyse critique des performances observées.

Enfin, le chapitre cinq permet de synthétiser les principaux résultats, de discuter des implications de l'étude et de proposer des pistes d'amélioration pour les travaux futurs.

Chapitre 2

Revue de littérature

La génération automatique de tests unitaires dans les systèmes orientés objet a fait l'objet de recherches approfondies au cours des dernières décennies, reflétant une diversité d'approches et de technologies. Cette revue de littérature vise à dresser un panorama des principales contributions dans ce domaine, en mettant en lumière les différentes méthodologies, outils, et avancées qui ont marqué cette recherche.

2.1 Génération guidée par la recherche (Search-Based Software Testing, SBST)

Les techniques de test logiciel basées sur la recherche (SBST) transforment le test en un problème d'optimisation pour générer des cas de test unitaires. L'objectif du SBST est de créer des suites de test optimales qui améliorent la couverture du code et révèlent efficacement les erreurs du programme en utilisant des algorithmes évolutionnaires, des algorithmes génétiques ou d'autres méthodes heuristiques pour explorer l'espace des problèmes. Cette approche montre un potentiel pour réduire le nombre de cas de tests nécessaires tout en maintenant un niveau de détection fiable des erreurs.

2.1.1 Travaux clés

Fraser et Arcuri (2014) [5] ont réalisé une revue complète des techniques SBST, mettant en évidence l'efficacité des algorithmes génétiques dans la génération de tests unitaires. Leurs travaux ont démontré que ces méthodes peuvent atteindre une couverture de code élevée, en particulier avec des outils comme EvoSuite, qui a été largement adopté dans la recherche et l'industrie.

Fraser et Arcuri [6] ont développé EvoSuite, un outil de génération automatique de tests unitaires pour Java basé sur des algorithmes génétiques. L'objectif principal était de surmonter les limites des outils existants (tels que JUnit-QuickCheck [7]) en générant des suites de test entières plutôt que des tests individuels.

EvoSuite [8] utilise des algorithmes génétiques pour faire évoluer des populations de suites de tests. Il applique des algorithmes évolutionnaires, inspirés de la sélection naturelle, pour

modifier et améliorer progressivement les suites de test au fil des générations. Chaque suite est évaluée en fonction de sa capacité à couvrir différentes parties du code, y compris les branches, les conditions, et les exceptions.
Le processus de sélection favorise les suites qui maximisent la couverture de code et détectent les anomalies. Les tests sont mutés et croisés pour explorer différentes combinaisons et améliorations.
Panichella, A., F.M. Kifetew, et P. Tonella (2017) [9] ont introduit DynaMOSA (Dynamic Many-Objective Sorting Algorithm) qui est une extension de l'algorithme MOSA [10], conçue pour améliorer l'efficacité de la génération de tests unitaires en se concentrant dynamiquement sur les objectifs non couverts au fur et à mesure du processus de génération. L'approche reformule le problème de la génération de tests comme un problème d'optimisation multi-objectifs dynamiques, où les objectifs (par exemple, les branches à couvrir) évoluent en fonction des résultats intermédiaires des tests. L'accent est mis sur la priorisation adaptative des objectifs non atteints, ce qui permet de maximiser la couverture plus rapidement et de manière plus ciblée.

Les approches basées sur la recherche offrent souvent une meilleure couverture qu'un simple test aléatoire, car elles utilisent un mécanisme d'optimisation guidé par une fonction coût. Toutefois, elles requièrent une définition de coût adaptée et peuvent être plus coûteuses en temps de calcul.

2.2 Génération aléatoire (Random Testing)

Les techniques de génération de tests randomisés se basent sur la création de séquences aléatoires d'actions et d'entrées pour explorer différents chemins dans un programme. Contrairement aux méthodes basées sur des modèles ou des analyses statiques, ces techniques misent sur l'exécution effective du code avec des valeurs d'entrée aléatoires pour découvrir des comportements inattendus et des bogues potentiels.

2.2.1 Travaux clés

C. Pacheco et M-D. Ernst (2007) [11] ont introduit Randoop, un outil basé sur la génération de tests aléatoires, qui s'améliore dynamiquement en excluant les cas de test non pertinents. Leurs recherches ont montré que Randoop pouvait détecter des bogues complexes avec une faible intervention humaine. L'approche utilise un mécanisme de feedback pour améliorer progressivement la génération de tests. Si une séquence de tests provoque une erreur, Randoop

la conserve et s'appuie dessus pour générer de nouvelles séquences potentiellement plus efficaces.

Le test aléatoire pur peut rapidement obtenir une couverture basique, mais peut peiner à couvrir des scénarios complexes nécessitant des valeurs très spécifiques ou des séquences d'appels précises. Pour contourner cela, les chercheurs et développeurs introduisent souvent du feedback ou de l'apprentissage pour affiner la génération.

2.3 Outils alternatifs à EvoSuite et Randoop

Parmi les outils de génération automatique de tests en boîte blanche, jPET (Albert, E., et al.) [12] se distingue par son intégration dans l'environnement Eclipse et sa capacité à générer des cas de test directement à partir du bytecode Java. Contrairement à EvoSuite et Randoop, qui génèrent des tests principalement à partir du code source ou via des approches évolutives ou aléatoires, jPET mise sur l'analyse structurelle des chemins d'exécution et sur la rétro-ingénierie pour produire des tests exploitables au niveau du code source.
Bien qu'il ne soit pas exclusivement dédié aux tests unitaires, son fonctionnement ciblé au niveau des classes et des méthodes en fait un outil pertinent pour la génération de tests unitaires guidés par la structure du programme, notamment dans les phases de développement précoce.

2.4 Génération guidée par l'exécution symbolique (Symbolic Execution)

N. Tillmann et J. de Halleux (2008) [13] ont introduit PEX, une approche basée sur une exécution symbolique et cible principalement C#/.NET. L'exécution symbolique consiste à parcourir les chemins possibles du code en traitant les valeurs d'entrées comme des symboles. Les chemins conditionnels (if, while, etc.) génèrent des contraintes (p. ex. $x > 0$). Un solveur est utilisé pour trouver les valeurs concrètes satisfaisant ou contournant ces contraintes.
L'exécution symbolique est plus systématique et déterministe, mais peut rencontrer des difficultés d'échelle (explosion combinatoire des chemins), et nécessite un solveur de contraintes. Elle est souvent combinée à des techniques de tests aléatoires ou de recherche ("concolic testing") pour en limiter les coûts.

2.5 Génération guidée par modèles ou contrats (Model-Based / Contract-Based)

Dans une approche Model-Based, on part de modèles UML, de machines à états ou de spécifications formelles. On peut construire un modèle UML (machine à états, diagramme de

séquence, etc.) décrivant le comportement d'une classe (ou d'un petit ensemble de classes) de façon relativement locale. À partir de ce modèle, on génère ensuite les cas de test qui vont invoquer les méthodes de cette classe (dans un certain ordre, avec certaines valeurs) pour parcourir tous les états du modèle. Les tests sont alors générés en explorant systématiquement le modèle [14].

Dans une approche Contract-Based (aussi appelée "Design by Contract"), on utilise les invariants, pré/post-conditions (par exemple, JML pour Java ou contrats .NET). Les outils de test peuvent alors générer des entrées qui respectent (ou violent) les contrats spécifiés.

2.6 Avancées avec les Grands Modèles de Langage

Plus récemment, l'introduction des grands modèles de langage (LLM) a ouvert de nouvelles perspectives pour la génération de tests unitaires. Les LLM, tels que GPT-3/3.5/4 (OpenAI), Codex (OpenAI), AlphaCode (DeepMind), ou encore les modèles intégrés dans GitHub Copilot (Microsoft), ont démontré ces dernières années leur capacité à traiter du code et à en produire de nouvelles portions pertinentes. Leur utilisation pour la génération de tests (tests unitaires, d'intégration, etc.) est rapidement devenue un sujet de recherche et d'expérimentation industrielle.

Tufano, M., et al. (2020) [15], de Microsoft, développent AthenaTest, un outil de génération de cas de tests unitaires automatisés qui utilise le Transformer BART [16]. Il a été évalué sur cinq projets Defects4j, générant 25 000 cas de test réussis couvrant 43,7 % des méthodes focales avec seulement 30 tentatives. Les auteurs affirment qu'une enquête auprès des développeurs professionnels a montré une préférence écrasante pour la lisibilité, la compréhensibilité et l'efficacité des tests générés par AthenaTest, par rapport à ceux d'EvoSuite. Bien que cet outil génère des cas de test lisibles et compréhensibles, leur qualité peut varier. Certains tests peuvent ne pas couvrir tous les scénarios ou ne pas détecter certains défauts.

Chen, M. et al. (2021) [17] présentent Codex, un modèle dérivé de GPT spécialisé dans le code, et l'évaluent sur des tâches de complétion et de génération de code (ex. génération de fonctions, correction de bogues). Ils évoquent la génération de tests unitaires comme cas d'usage, montrant que le modèle peut produire du code de test plausible sans garantie formelle.

Yuan, Z. et al. (2024) [18] étudient la capacité de ChatGPT à générer automatiquement des tests unitaires. Malgré une bonne couverture et une lisibilité proche de tests écrits à la main, les tests fournis par ChatGPT présentent encore des problèmes d'exactitude (erreurs de

compilation, assertions incorrectes). Pour y remédier, ils proposent ChatTester, un système qui améliore de façon itérative les tests initialement générés par ChatGPT. Selon leurs résultats, ChatTester produit davantage de tests compilables et plus d'assertions correctes que ChatGPT seul.

Sapozhnikov, A., et al. [19] ont mis en place TestSpark, un plugin open source pour IntelliJ IDEA conçu pour faciliter la génération de tests unitaires directement dans l'environnement de développement. Il s'appuie à la fois sur des approches traditionnelles de génération de tests automatisés et sur les grands modèles de langage, en établissant un cycle de rétroaction entre le LLM et l'IDE. Contrairement à des outils comme EvoSuite ou AthenaTest, souvent complexes ou limités à une seule technique, TestSpark met l'accent sur la concrète utilisabilité, permettant aux développeurs de modifier, exécuter et intégrer facilement les tests générés dans leurs projets. Son architecture extensible permet aussi d'ajouter de nouvelles stratégies de génération avec peu d'effort.

Tous ces travaux suggèrent que les LLM pourraient non seulement automatiser la génération de tests mais aussi produire des cas de test plus diversifiés que certaines approches automatisées traditionnelles basées sur des heuristiques ou des algorithmes génétiques, en s'appuyant sur leur capacité à exploiter des informations contextuelles issues du code et des connaissances acquises sur de larges corpus.

Ainsi, à partir des constats établis dans cette revue de littérature, le chapitre suivant présente les approches explorées et le cadre méthodologique adopté pour évaluer l'efficacité des LLM dans la génération de tests unitaires, en détaillant le protocole expérimental, les outils utilisés et les métriques retenues.

Chapitre 3

Approches explorées et cadre méthodologique

Dans les chapitres précédents, nous avons posé le contexte général de cette étude et défini les objectifs que nous souhaitions atteindre. Afin de répondre à ces objectifs, plusieurs approches ont été envisagées, testées et comparées. Le présent chapitre vise donc à décrire la démarche méthodologique suivie, en distinguant clairement les différentes pistes explorées de l'approche finalement adoptée.

Nous commencerons par présenter, dans la section 3.1, l'ensemble des outils et de l'environnement d'expérimentation utilisés. Il s'agit notamment de décrire les ressources logicielles et matérielles mobilisées, ainsi que les protocoles d'évaluation généraux qui serviront de base à nos comparaisons.

La section 3.2 sera ensuite consacrée à l'exposition des approches explorées. Nous détaillerons les principes de fonctionnement de chaque méthode, en soulignant leurs avantages, leurs limites, les données utilisées et les raisons pour lesquelles elles ont été retenues ou écartées au cours de cette étude.

Enfin, la section 3.3 traitera de la méthodologie adoptée. Nous y expliciterons plus en profondeur le choix de l'approche finale, ses fondements théoriques et pratiques, ainsi que les paramètres spécifiques de mise en œuvre. Ce cadre méthodologique servira de référence pour l'expérimentation et l'interprétation des résultats et la discussion qui seront développées au chapitre 4.

Cette structuration nous permettra d'abord d'établir une comparaison solide entre les différentes pistes considérées, puis de justifier notre démarche finale de manière claire et argumentée.

3.1 Outils d'expérimentation

3.1.1 Mistral

Pour la réalisation des différentes approches nécessitant l'utilisation de grands modèles de langage (LLM), il est crucial de faire un choix judicieux. En particulier, lorsque les ressources sont limitées, il devient essentiel de sélectionner un modèle qui offre un bon équilibre entre performance et exigences matérielles. Parmi les nombreuses options disponibles, le modèle

Mistral 7B [20] se distingue comme une option particulièrement pertinente pour répondre à ces besoins. Voici pourquoi :

3.1.1.1 Performance supérieure

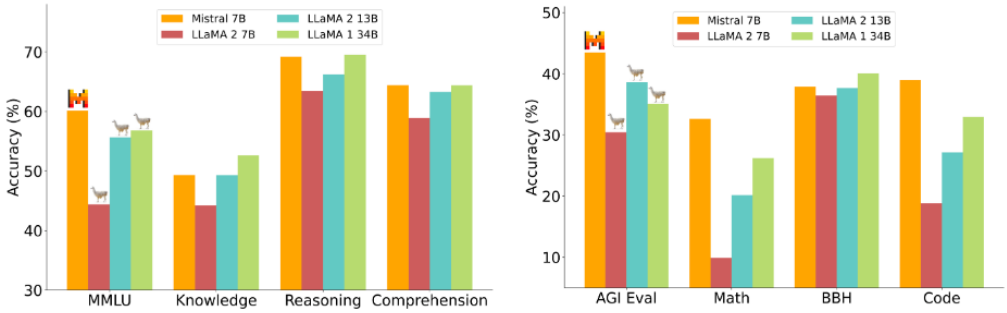

Figure 1 : Performances de Mistral 7B [20]

Le Mistral 7B, avec ses 7 milliards de paramètres, offre des performances remarquables qui surpassent même des modèles de plus grande envergure comme Llama 2 13B sur tous les benchmarks évalués, et se distingue également par sa capacité à comprendre et interpréter le code, y compris des langages comme Java, Python, et JavaScript.

Il utilise des mécanismes pour améliorer la vitesse d'inférence et traiter efficacement des séquences plus longues tels que :

- Attention par fenêtre coulissante

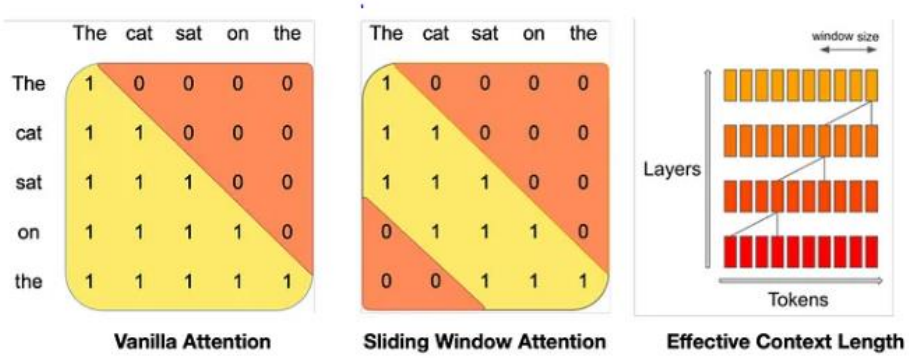

Figure 2 : L'attention par fenêtre coulissante de Mistral 7B [20]

L'attention par fenêtre coulissante (sliding window attention) est une variation du mécanisme d'attention utilisé principalement pour optimiser le traitement des séquences longues et pour réduire la complexité computationnelle associée aux modèles de Transformer traditionnels. Le mécanisme d'attention standard dans un Transformer calcule les scores d'attention en considérant chaque paire de tokens dans l'entrée, ce qui entraîne une complexité quadratique

par rapport à la longueur de la séquence ($O(n^2)$). Pour les séquences très longues, cela peut devenir très coûteux en termes de calcul et de mémoire. Pour pallier ce problème, l'attention par fenêtre coulissante limite l'attention à une "fenêtre" de taille fixe autour de chaque token. Cela signifie que pour un token donné, le modèle ne calcule l'attention que pour les tokens à l'intérieur de cette fenêtre (par exemple, les 128 tokens précédents et suivants), plutôt que pour tous les tokens de la séquence.

Mistral est donc entraîné avec une longueur de contexte de 8 000 et une taille de cache fixe, avec une durée d'attention théorique de 128 000 jetons.

- GQA (Grouped Query Attention)

  Le GQA repose sur l'idée de regrouper les requêtes dans le mécanisme d'attention avant de calculer les scores d'attention. Voici comment cela fonctionne généralement :

  Regroupement des Requêtes :

  Les requêtes (c'est-à-dire les représentations des tokens qui cherchent à savoir où "regarder" dans le reste de la séquence) sont regroupées en plusieurs sous-ensembles. Cela peut se faire selon divers critères, tels que la proximité dans la séquence ou des caractéristiques partagées.

  Calcul de l'Attention :

  Au lieu de calculer l'attention pour chaque requête individuellement par rapport à toutes les clés de la séquence, l'attention est calculée pour chaque groupe de requêtes. Cela réduit le nombre de calculs d'attention nécessaires.

  Le GQA permet donc à Mistral une inférence plus rapide et une taille de cache inférieure.

3.1.1.2 Accessibilité

Mistral 7B est conçu pour être utilisé efficacement même sur des machines avec des ressources limitées. Il peut être entraîné et utilisé sur une machine équipée d'un GPU avec 80 Go de RAM, ce qui le rend idéal pour des environnements avec des contraintes matérielles strictes. De plus, grâce à des techniques comme QLoRA (*Quantized Low Rank Adaptation*) [21], qui combine quantisation en 4 bits *et Low-Rank Adaptation*, il est possible de fine-tuner le modèle sur des GPUs de capacité limitée tout en maintenant une haute qualité des résultats.

Notons, enfin, que Mistral 7B est disponible en open source et peut être facilement téléchargé sur plusieurs plateformes comme Hugging Face [22], rendant son utilisation accessible à un large public. Cette accessibilité permet aux développeurs et chercheurs de bénéficier des

capacités avancées de Mistral 7B sans coûts supplémentaires liés aux licences ou à l'infrastructure coûteuse.

3.1.2 JUGE

Pour évaluer l'efficacité des tests unitaires générés, nous avons sélectionné JUGE [23], une infrastructure robuste spécialement conçue pour tester et évaluer les performances des outils de génération de tests en Java. Cette plateforme permet une évaluation précise grâce à ses capacités de couverture structurelle et d'analyse de mutation, offrant ainsi une mesure fiable des capacités de détection de défauts des suites de tests générées. Il a été utilisé dans plusieurs éditions des compétitions liées aux tests unitaires automatisés, co-organisées avec des ateliers tels que le Search-Based Software Testing Workshop (SBST) [24] .

Cette approche d'évaluation suit des principes largement reconnus dans l'industrie du logiciel et la recherche en ingénierie logicielle, qui recommandent l'utilisation de benchmarks standardisés et de méthodes d'analyse rigoureuses pour comparer objectivement différents outils de génération de tests unitaires. Les benchmarks fournissent un cadre commun qui favorise la comparabilité, la reproductibilité et l'objectivité des évaluations, tandis que l'analyse statistique permet d'interpréter les résultats de manière fiable.

3.1.2.1 Fonctionnalités et Capacités de JUGE

JUGE (JUnit Generation Benchmarking Infrastructure) propose plusieurs fonctionnalités essentielles pour une évaluation complète des outils de génération de tests. Il automatise l'exécution des suites de tests générées contre diverses bases de code Java, permettant ainsi une évaluation pratique de leur performance dans des environnements réels. Dans JUGE, chaque suite de tests générée est d'abord compilée. Les tests non compilables sont automatiquement identifiés et exclus des phases suivantes. Pour les tests compilables, JUGE exécute une instrumentation via JaCoCo **[25]** pour mesurer la couverture de lignes et de branches, et utilise PIT **[26]** pour calculer le score de mutation. Les résultats sont ensuite collectés dans des rapports standardisés, incluant des statistiques sur les tests irréguliers et non compilables, ainsi que la couverture des lignes et des branches, et l'analyse des mutations. Ces éléments offrent un feedback précieux pour le raffinement continu des outils de génération de tests.

Plus précisément, JUGE évalue la qualité des tests unitaires à travers plusieurs métriques clés :

- Couverture de lignes : Mesure le pourcentage de lignes de code exécutées par les tests. Une couverture élevée indique que les tests s'exercent sur une grande partie du code.
- Couverture de branches : Évalue le pourcentage de branches conditionnelles (résultants des structures de contrôle du langage) couvertes par les tests. Une couverture élevée montre que les tests vérifient toutes les conditions logiques possibles**.**
- Analyse des mutations : Consiste à créer des versions modifiées du code (mutants) pour vérifier si les tests peuvent détecter ces modifications. Ces dernières simulent des erreurs ou des fautes possibles, comme changer un opérateur logique, modifier une constante, ou altérer une condition. L'objectif principal de cette technique est de vérifier si les tests existants sont capables de détecter ces changements. Si un test échoue à identifier un mutant (c'est-à-dire si le mutant survit), cela indique que le test pourrait ne pas être assez robuste pour détecter des erreurs similaires dans un contexte réel. Le score de mutation reflète le pourcentage de mutants détectés, indiquant ainsi la capacité des tests à identifier les erreurs. JUGE optimise cette analyse en se basant sur la couverture de lignes pour exécuter uniquement les tests concernés par les lignes de code modifiées.

Pourquoi ces métriques ?

Les métriques de couverture de lignes, de branches, et le score de mutation sont choisies parmi d'autres indicateurs pour leur pertinence et leur complémentarité dans l'évaluation de la qualité des tests logiciels.

- **Complémentarité :** Utilisées ensemble, elles couvrent différents aspects de la qualité des tests. Par exemple, un code avec une couverture de lignes élevée peut toujours manquer de branches testées, et un score de mutation peut montrer que les assertions des tests sont faibles.
- **Validation empirique :** Des études démontrent que ces métriques sont corrélées à la capacité des tests à détecter des bogues réels et à améliorer la qualité globale du logiciel [27] [28].
- **Normes industrielles :** Elles sont largement acceptées dans les outils (comme JaCoCo [25] pour la couverture et PIT [26] pour le test de mutation) et les environnements professionnels, facilitant leur adoption et leur compréhension.

3.1.2.2 Architecture de JUGE

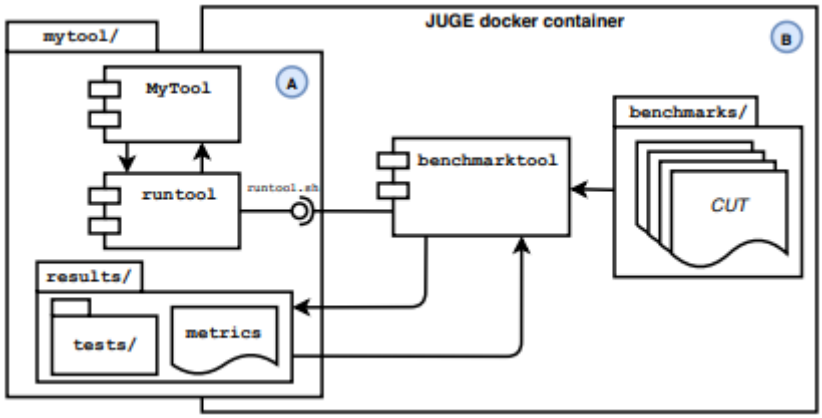

Figure 3 : Architecture de JUGE [23]

L'architecture de JUGE s'articule autour de plusieurs composants clés, chacun jouant un rôle spécifique dans le processus d'évaluation et d'exécution des tests. Ces composants incluent :

- **Conteneur Docker (B) :** JUGE fonctionne dans un conteneur Docker, assurant ainsi l'isolation de l'exécution des tests et des générateurs de tests par rapport au système hôte.
- **Runtool Adapter :** Cet adaptateur encapsule les appels spécifiques au générateur de tests MYTOOL et offre une interface au benchmarktool.
- **Benchmarktool :** Responsable de l'orchestration de l'évaluation du générateur de tests unitaires.
- **Communication via un Dossier Commun (A) :** La communication entre l'hôte et le conteneur Docker de JUGE se fait via un dossier commun monté dans la structure de fichiers de l'image Docker. Ce dossier contient :
    - Binaries : Les exécutables du générateur de tests unitaires.
    - Runtool Adapter : L'adaptateur runtool.
    - Résultats : Les tests générés, les métriques, et les résultats de l'analyse statistique sont sauvegardés dans un sous-dossier (results/) accessible depuis l'hôte. Cela permet de récupérer facilement les résultats pour une analyse ultérieure.
- **Classes sous Test (CUT) et Configuration :** Les classes à tester et le fichier de configuration correspondant sont stockés dans le conteneur Docker (benchmarks/). Cela permet de réutiliser le même conteneur pour évaluer plusieurs outils en montant simplement différents dossiers contenant les générateurs de tests unitaires et leurs adaptateurs.

3.1.2.3 Adaptation de JUGE pour l'évaluation de nos approches

Pour adapter JUGE à un nouveau générateur de tests unitaires, les étapes ci-dessous ont été suivies. L'objectif étant d'intégrer notre nouvel outil UtGen développé, en adaptant la partie MYTOOL de JUGE.

a. Compréhension de l'Existant

JUGE fournit un exemple d'intégration pour un générateur de tests unitaires, appelé MYTOOL. Ce dernier est conçu pour appeler les outils de génération de tests unitaires avec les entrées suivantes :

- **--testclass :** Spécifie la classe Java à tester
- **--junit-output-dir :** Spécifie le répertoire de sortie où les fichiers de tests JUnit générés seront enregistrés
- **--junit-package-name :** Spécifie le nom du package dans lequel les tests JUnit générés seront placés.

b. Implémentation de ut_llm.py dans Mytool

Nous avons ainsi mis en place le script Python ut_llm.py qui accepte les mêmes arguments d'entrée utilisés dans MYTOOL. Le script ut_llm est un outil flexible qui automatise le processus de génération de tests unitaires pour les approches Fine-tuning, Zero-shot ou Few-shot.

3.2 Approches explorées

Dans cette section, nous explorons diverses méthodes utilisées pour améliorer la performance des modèles, notamment le *fine-tuning*, le *zero-shot prompting* et le *few-shot prompting*, en détaillant les principes généraux de chaque approche ainsi que leurs expérimentations spécifiques avec Mistral.

Cependant, avant de nous concentrer sur les approches principales de fine-tuning et de prompting, nous avons exploré diverses méthodes pour améliorer la capacité de notre modèle à comprendre le code source. Ces efforts incluaient notamment l'utilisation des arbres syntaxiques abstraits (AST), une technique largement utilisée pour représenter la structure syntaxique du code. Cette section va donc d'abord détailler cette tentative exploratoire, les motivations derrière leur sélection, ainsi que les raisons pour lesquelles elle n'a pas été retenue dans la suite de l'étude.

3.2.1 Approche exploratoire initiale : Utilisation des Arbres Syntaxiques Abstraits (AST)

Dans la quête d'améliorer la capacité de notre modèle à comprendre le sens sémantique du code, nous avons d'abord exploré l'utilisation des Arbres Syntaxiques Abstraits (AST). Les AST sont une représentation structurée et hiérarchique du code source d'un programme, où chaque nœud de l'arbre représente une construction dans le langage de programmation telle que les boucles, les conditions, et les déclarations de fonctions, etc. Les AST sont cruciaux pour de nombreuses applications en ingénierie logicielle et en traitement automatique du code, car ils capturent à la fois la structure et les éléments syntaxiques du code d'une manière qui peut être facilement analysée et manipulée par des programmes.

- **Structure des AST**

  Dans un AST, le programme est décomposé en ses composants élémentaires :

  Nœuds : Chaque nœud représente une structure ou une opération dans le code, telle qu'une déclaration de variable, une affectation, une opération mathématique, une structure de contrôle (if, loop), etc.

  Arêtes : Les connexions entre les nœuds représentent la relation syntaxique entre les différents éléments du code, comme la séquence d'opérations ou la hiérarchie de blocs de code.

  Exemple en Java : Voici une représentation simplifiée de l'AST pour ce code : la *figure 4* est un exemple de méthode Java pour additionner deux nombres et la *figure 5* représente son AST.

```
public int add (int x, int y){
        return x + y ;
}
```

Figure 4 : code source Java de la méthode add

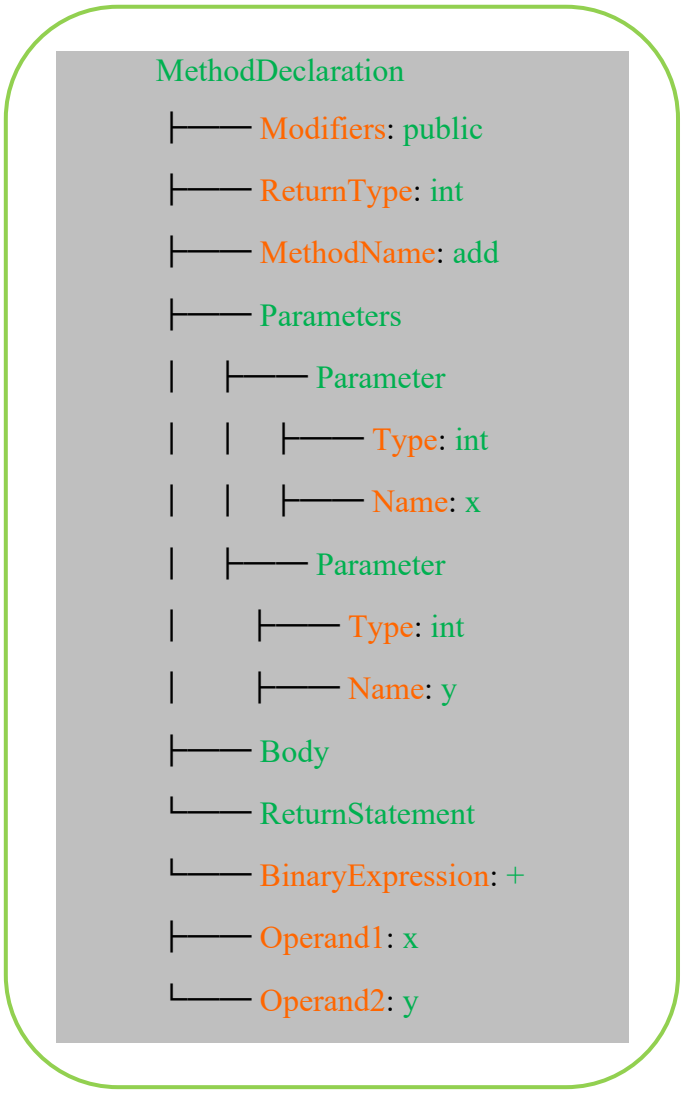

Figure 5 : Représentation de l'AST pour la méthode add

- **MethodDeclaration** : Définit une méthode avec ses modificateurs, son type de retour, son nom, ses paramètres, et son corps.
  - **Modifier** : Indique que la méthode est *public*.
  - **Type** : Le type de retour de la méthode est *int*.
  - **Name** : Le nom de la méthode est *add*.
  - **Parameters** : Liste des paramètres de la méthode.
    - ✓ **Parameter** : Chaque paramètre a un type *int* et un nom *x* et *y*.
- **Body** : Contient les instructions exécutées par la méthode
  - **ReturnStatement** : La méthode retourne le résultat d'une expression.
    - ✓ **BinaryExpression** : Représente l'opération binaire + (addition)
      - **Operand1** et **Operand2** : Les deux opérandes de l'expression, ici les identifiants x et y.

Pour chaque fragment de code dans notre ensemble de données, un AST a été généré en utilisant un parseur standard (JavaParser [29] qui est une bibliothèque spécifique à Java qui fournit un AST facile à manipuler pour l'analyse ou la transformation de code).
Les AST ont ensuite été transformés pour aligner avec les besoins de notre modèle, incluant l'extraction des nœuds significatifs et la normalisation des identificateurs. Cette transformation consiste à parcourir et filtrer l'AST pour ne conserver que les nœuds pertinents.
Cependant, un des principaux défis rencontrés avec l'utilisation des AST a été leur complexité et leur grande taille, surtout pour les bases de code volumineuses ou complexes. Cela rend difficile leur utilisation directe dans notre modèle d'apprentissage automatique car facilement ils deviennent trop grands pour être gérés efficacement par les mécanismes d'attention des modèles à base de Transformers.
Aussi, bien que les AST capturent la structure du code, ils ne capturent pas le contexte sémantique ou les intentions de programmation de manière qui soit immédiatement utile pour la génération de tests automatisés.
Pour surmonter ces limitations, nous avons exploré une alternative en utilisant Code2Vec [30], qui permet de transformer un AST en une représentation vectorielle de taille fixe. Cette approche offre l'avantage de réduire la complexité en simplifiant l'entrée pour le modèle.

- **Reduction des AST avec Code2Vec et limitations**

Code2vec [30] propose une méthode pour apprendre des représentations distribuées de fragments de code. Le modèle code2vec a été conçu pour transformer des extraits de code source en vecteurs denses, ce qui permet de représenter sémantiquement les fragments de code de manière compacte. L'approche se concentre sur l'extraction de caractéristiques à partir des arbres syntaxiques abstraits (AST) du code source. Spécifiquement, il extrait des chemins dans ces arbres et les utilise pour prédire le nom d'une méthode, servant ainsi de sommaire sémantique de ce que fait le code. Ces chemins extraits ont été encodés en utilisant un modèle d'embedding qui transforme chaque chemin en un vecteur dense de taille fixe. L'ensemble de ces vecteurs forme une représentation sémantique du fragment de code initial. Cette transformation permet d'utiliser les techniques d'apprentissage profond qui nécessitent une entrée de taille fixe, facilitant ainsi l'intégration de la sémantique du code dans les modèles d'apprentissage machine.
Cependant les limitations (perte d'informations) observées avec Code2Vec montrent que, bien que la représentation en vecteurs de taille fixe offre des avantages en termes de traitement, elle peut également conduire à une perte d'information, en particulier pour les fragments de code qui sont soit trop simples soit trop complexes par rapport à la taille du vecteur.

- **Conclusion**

Compte tenu de ces obstacles, nous avons décidé d'abandonner l'approche basée sur les AST. La taille et la complexité des AST rendent leur utilisation directe difficile dans un modèle d'apprentissage automatique, tandis que les techniques de réduction de dimension comme Code2Vec induisent une perte d'information qui impacte la qualité de la génération des tests. Ces limitations nous ont poussés à explorer d'autres représentations du code plus adaptées à notre problématique.

3.2.2 Fine-tuning

3.2.2.1 Généralités

a. Définition

Le *fine-tuning* [31] est une technique qui consiste à adapter un modèle de langage pré-entraîné sur un ensemble de données spécifique à une tâche particulière. Contrairement au *zero-shot* ou au *few-shot prompting*, où le modèle s'appuie sur ses connaissances générales ou sur quelques exemples, le *fine-tuning* implique un entraînement supplémentaire du modèle sur un ensemble de données spécifique à la tâche.

b. Processus de Fine-Tuning

- **Collecte des Données** *:*

Le but est de rassembler un ensemble de données représentatif de la tâche à accomplir. Les données doivent être bien annotées et de haute qualité pour garantir un *fine-tuning* efficace. Par exemple, pour un modèle de résumé de texte, collecter des paires de textes et de leurs résumés correspondants. Les données concrètement utilisées dans le cadre de cette étude sont présentées en détail dans la section 3.2.2.2.

- **Préparation des Données** :

Cette étape consiste à nettoyer et à structurer les données pour qu'elles soient compatibles avec le modèle de langage. Les données doivent être formatées de manière cohérente, avec des étiquettes claires et des exemples variés. C'est dans ce processus par exemple que l'on va supprimer les éléments non pertinents, normaliser les formats de texte, et structurer les données en ensembles de formation et de validation.

- **Entraînement du Modèle :**

L'entrainement va ajuster les poids du modèle en fonction des données spécifiques de la tâche. Il se base sur des algorithmes d'apprentissage supervisé pour mettre à jour les paramètres du modèle. On peut, par exemple, entraîner le modèle avec des itérations multiples en utilisant des techniques comme la descente du gradient pour minimiser l'erreur de prédiction.

- **Évaluation et Ajustement :**

Cette étape va évaluer la performance du modèle « fine-tuné » sur un ensemble de données de test et ajuster les hyperparamètres si nécessaire. On utilise des métriques de performance appropriées pour la tâche (par exemple, précision, rappel, F1-score). Dans le cas de résumé de texte, on peut par exemple comparer les résumés générés par le modèle aux résumés de référence et ajuster les paramètres d'entraînement pour améliorer les résultats.

c. Avantages

Le *fine-tuning* pourrait permettre d'augmenter considérablement la précision d'un modèle sur une tâche spécifique en adaptant ses paramètres aux particularités des données de cette tâche. Le modèle pourrait devenir plus performant pour des tâches spécifiques en se spécialisant dans les types de données et les exigences de cette tâche.
Grâce au *fine-tuning*, le modèle peut être régulièrement mis à jour avec de nouvelles données, afin d'améliorer continuellement ses performances.

d. Limitations

Le fine-*tuning* nécessite un ensemble de données annotées de haute qualité, ce qui peut être coûteux et chronophage à obtenir. D'un autre côté, l'entraînement des modèles nécessite des ressources informatiques importantes, notamment des GPU puissants et du temps de calcul. Notons que le modèle fine-tuné peut surapprendre et devenir trop spécialisé sur l'ensemble de données d'entraînement, perdant ainsi sa capacité à généraliser à des exemples qu'il n'a pas vus auparavant. Enfin, le processus de *fine-tuning* peut être complexe et nécessiter une expertise en apprentissage automatique pour sélectionner les bons hyperparamètres et éviter les problèmes courants comme le surapprentissage.

3.2.2.2 Expérimentation avec Mistral

a. Ensemble de données Method2test

- **Présentation**

Tableau 1 : Ensemble de données Methods2Test

Ensemble	Dépôts	Cas de test mappés
Entrainement	72,188	624,022
Validation	9,104	78,534
Test	10,093	78,388
Total	78,388	780,944

Pour entraîner et évaluer le modèle, un ensemble de données diversifié et représentatif est crucial. Dans ce projet, nous avons utilisé Method2test [32] qui est un ensemble de données supervisé open source, associant des cas de test JUnit à leurs méthodes focales correspondantes ou méthodes ciblées par le test (780 944 de paires test-méthode), extrait d'une variété étendue de dépôts de logiciels Java (91 385 de projets Java analysés) (voir *tableau 1*).

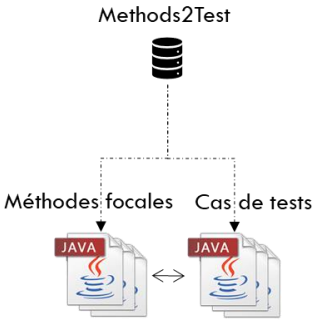

Figure 6 : L'ensemble Method2test

Pour assembler cet ensemble de données, les chercheurs [32] ont initialement examiné divers projets Java afin de recueillir des informations sur les classes et méthodes, y compris leurs métadonnées associées (nom de la classe, type de classe, annotations…). Ils ont ensuite procédé à l'identification des classes de test et de leurs classes focales correspondantes. Chaque cas de test au sein d'une classe de test a été associé à la méthode focale correspondante, aboutissant à la création d'une collection de cas de test clairement mappés (*figure 6*).

Les classes de test sont définies comme telles lorsqu'elles contiennent au moins une méthode annotée avec @Test. Cette annotation sert à indiquer à JUnit que la méthode peut être exécutée en tant que cas de test.

- **Définition de la méthode focale**

Les méthodes focales désignent les méthodes qui sont ciblées lors des tests. Pour chaque cas de test, c'est-à-dire une méthode au sein d'une classe de test marquée par l'annotation @Test, les auteurs ont cherché à identifier la méthode focale appropriée dans la classe focale. Pour y parvenir, ils ont appliqué des heuristiques basées principalement sur la correspondance des noms. Conformément aux pratiques courantes de nomenclature, les noms des cas de test reflètent souvent ceux des méthodes focales correspondantes. Par conséquent, la première heuristique appliquée cherche à associer les cas de test aux méthodes focales dont les noms sont identiques, après avoir éliminé les préfixes ou suffixes liés au test.

b. Préparation de Method2test

- **Choix de la structure d'entrée du modèle :**

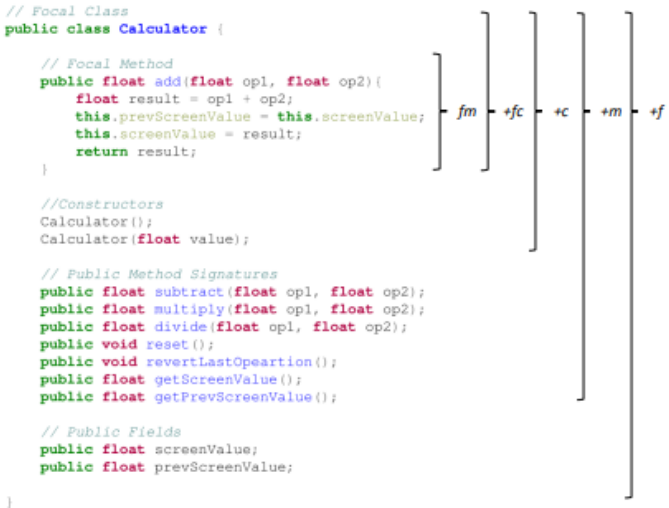

Figure 7 : Le contexte focal ou spécifique

Dans le cadre du développement de notre modèle pour la génération de cas de test, nous avons choisi de former le modèle avec une entrée structurée, *figure 7*, autour du contexte focal (contexte spécifique) enrichi, spécifiquement la configuration "fm+fc+c+m+f". Cette méthode intègre non seulement le code source de la méthode sous test (fm), mais également le nom de la classe où elle est définie (fc), les signatures des constructeurs (c), les autres méthodes publiques (m), et les champs publics (f) de cette classe. Cette stratégie vise à fournir au modèle une compréhension contextuelle de la méthode focale, ce qui devrait permettre une meilleure génération de tests pertinents et fonctionnels. En effet, les auteurs [32] ont constaté qu'entraîner un modèle avec le contexte de sa classe peut effectivement améliorer la fonction perte de validation.

- **Restructuration des Données Methods2Test :**

Dans le cadre de notre utilisation de l'ensemble de données Methods2Test pour la formation de modèles de génération de tests unitaires, nous avons identifié une limitation structurelle significative dans la manière dont les cas de test étaient initialement organisés. Dans l'ensemble de données original, les cas de test associés à une méthode focale spécifique n'étaient pas systématiquement regroupés ensemble. Il n'était pas rare de trouver des cas de test pour une même méthode focale dispersés à travers l'ensemble des données. Cette dispersion pouvait potentiellement limiter la capacité du modèle à apprendre efficacement de toutes les variations de tests possibles pour une méthode donnée, car le model pourrait ne pas clairement percevoir la relation entre ces cas de test et leur méthode focale commune.

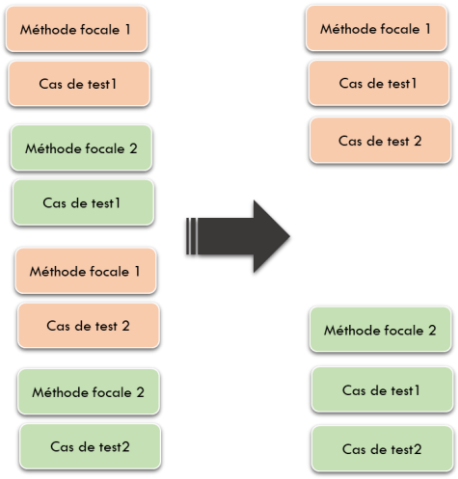

Figure 8 : Réorganisation de Method2test

Pour surmonter ce défi et optimiser l'entraînement de notre modèle, nous avons procédé à une réorganisation des données (*figure 9*). Nous avons regroupé tous les cas de test correspondant à une même méthode focale dans une seule et même entrée en appliquant l'heuristique basée sur la correspondance des noms. Cette restructuration vise à fournir au modèle un contexte plus riche et plus intégré, ce qui est crucial pour les approches d'apprentissage profond qui dépendent fortement de la compréhension du contexte et de la généralisation à partir d'exemples complets.

Cette réorganisation des données dans Methods2Test n'est pas seulement une modification technique; elle constitue une amélioration stratégique qui aligne l'ensemble de données avec les objectifs de notre modèle d'apprentissage pouvant potentiellement améliorer la capacité du modèle à générer des tests unitaires efficaces et variés pour les systèmes orientés objet. Cette

approche pourrait renforcer la pertinence pratique de la génération automatique de tests unitaires, contribuant à une meilleure assurance qualité dans le développement logiciel.

c. Fine-Tuning des Modèles Mistral avec l'API Mistral et LoRA

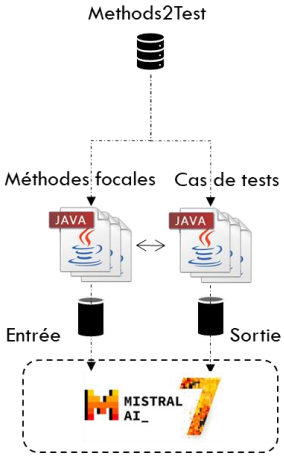

Figure 9 : Fine-tuning avec l'API Mistral

Dans le cadre de notre expérimentation avec Mistral pour la génération de cas de test, nous avons mis en place un fine-tuning des modèles Mistral en utilisant la base de code Mistral-fine-tuné, optimisée pour un fine-tuning performant et économe en mémoire. Ce fine-tuning s'appuie sur LoRA (Low-Rank Adaptation) [33], une méthode qui gèle la majorité des poids du modèle pré-entraîné et ajuste seulement 1 à 2 % de poids supplémentaires sous forme de perturbations matricielles de faible rang. Cette approche permet d'adapter le modèle de manière ciblée et à un coût abordable, réduisant les besoins en ressources tout en préservant la performance.

Pour réaliser ce fine-tuning, nous utilisons l'API de Mistral disponible sur La Plateforme, qui facilite l'accès aux modèles Mistral, qu'ils soient open-source ou commerciaux, sans nécessiter d'infrastructure GPU. En plus des options de personnalisation des paramètres d'entraînement, cette API permet d'utiliser directement leurs services pour l'inférence, offrant donc une solution intégrée pour le fine-tuning et l'application des modèles dans un environnement de production. Bien que Mistral-fine-tuné soit également disponible en open source pour des personnalisations avancées, l'API simplifie les étapes d'intégration et de réglage, ce qui la rend idéale pour notre expérimentation et pour l'utilisation continue des modèles fine-tunés à moindre coût.

3.2.3 Few-Shot Prompting

3.2.3.1 Généralités

a. Définition

Le *few-shot prompting* [34] est une technique dans laquelle un modèle de langage accomplit une tâche en s'appuyant sur un petit nombre d'exemples fournis dans le prompt. Contrairement au *zero-shot prompting*, cette méthode offre des exemples spécifiques de la tâche à accomplir, permettant ainsi au modèle de comprendre et de reproduire la structure avec le format attendu des réponses.

b. Structure Générale d'un Prompt

Pour créer un prompt efficace en few-shot prompting, il est crucial de structurer le message d'entrée de manière claire et organisée. Voici les éléments clés à inclure :

- **System Message (Facultatif mais Recommandé) :**

Il permet de définir le contexte ou le comportement global attendu du modèle.
Il guide le modèle sur la manière dont il doit répondre. Cela peut inclure le ton, le style ou le rôle que le modèle doit jouer. Par exemple : "Vous êtes un assistant qui, dans cet exercice, doit répondre en inversant les valences habituelles des phrases."

- **Instruction Spécifique (Prompt Principal) :**

Le prompt principal décrit précisément la tâche ou la question à laquelle vous voulez que le modèle réponde, en s'appuyant sur les exemples fournis.
En cela, il doit être clair et concis pour maximiser la compréhension du modèle. Cette section est cruciale car elle détermine l'action que le modèle doit effectuer.

Exemple :

  - "C'est génial !" // Négatif
  - "C'est mauvais !" // Positif
  - "Wow ce film était génial !" // Négatif
  - "Quel horrible spectacle !" // (il devrait répondre Positif)

- **Contexte Additionnel (Facultatif) :**

Le contexte additionnel fournit des informations supplémentaires pour orienter la réponse du modèle. Il est particulièrement utile pour des tâches nécessitant un contexte spécifique, comme des spécifications détaillées ou des exemples supplémentaires. Cela aide le modèle à mieux comprendre le contexte ou les détails de la tâche. Par exemple : "Les étiquettes de sentiment

sont délibérément mal attribuées pour tester la capacité du modèle à suivre des formats spécifiques sans se fier aux émotions typiques des phrases."

Le *few-shot prompting*, en fournissant quelques exemples concrets, permet d'améliorer la précision et la pertinence des réponses générées par le modèle, tout en offrant une flexibilité et une adaptabilité significatives. Cependant, il reste crucial de bien choisir et structurer ces exemples pour maximiser l'efficacité de cette technique.

c. Avantages

Le modèle peut fournir des réponses plus précises en se basant sur les exemples donnés, réduisant ainsi le risque de réponses incorrectes ou hors sujet. Par ailleurs, avec quelques exemples, le modèle peut rapidement apprendre et s'adapter à de nouvelles tâches sans nécessiter un grand ensemble de données d'entraînement. Cette approche confère au modèle la capacité de s'adapter à une variété de tâches en fournissant simplement quelques exemples pertinents, ce qui est particulièrement utile pour des tâches complexes ou spécifiques.

d. Limitations

Bien que le *few-shot prompting* puisse améliorer la précision par rapport au *zero-shot prompting*, il peut aussi rencontrer des difficultés pour généraliser à des contextes ou des tâches très différentes de celles des exemples fournis. Par ailleurs, le temps d'inférence peut être plus long en raison de la complexité accrue des prompts avec plusieurs exemples, ce qui peut ralentir le processus de réponse du modèle.

3.2.3.2 Expérimentation avec Mistral

Dans cette partie nous décrivons comment nous avons utilisé l’approche *few-shot prompting* pour générer des tests unitaires.

a. Few-shot prompting avec le contexte global

```
### System Message:
As an assistant, I need to help a Java developer write a unit test for
all public methods of the given class using JUnit 5.

### Instruction:
Generate comprehensive JUnit 5 unit tests for all public methods of the given
class,
ensuring maximum code coverage by thoroughly testing all possible scenarios and
edge cases.

### Requirements:
- Do not call private methods: Private methods should never be directly invoked in
unit tests.
- Do not access private variables: Private variables in the class should not be
accessed or manipulated directly in the tests.
- Never Use mock.
- Follow the AAA pattern: Structure each test according to the Arrange (set up the
test context), Act (execute the action being tested), and Assert (verify the
result) pattern.
- Cover critical use cases: Ensure that the tests cover key scenarios, including
normal cases, edge cases, and error scenarios.
- Import all necessary functionalities: Make sure to import all classes and
functions that are required to perform the tests.
- Never instantiate an abstract class.
- Never test private method.

### Java class with all public methods:
<example_class>

### Unit tests
<example_tests>

### Instruction:
Generate comprehensive JUnit 5 unit tests for all public methods of the given
class,
ensuring maximum code coverage by thoroughly testing all possible scenarios and
edge cases.

### Requirements:
- Do not call private methods: Private methods should never be directly invoked in
unit tests.
- Do not access private variables: Private variables in the class should not be
accessed or manipulated directly in the tests.
- Never Use mock.
- Follow the AAA pattern: Structure each test according to the Arrange (set up the
test context), Act (execute the action being tested), and Assert (verify the
result) pattern.
- Cover critical use cases: Ensure that the tests cover key scenarios, including
normal cases, edge cases, and error scenarios.
- Import all necessary functionalities: Make sure to import all classes and
functions that are required to perform the tests.
- Never instantiate an abstract class.
- Never test private method.

### Java class with all public methods:
<class_to_test>

### Unit tests
```

Figure 10 : prompt du few-shot contexte global

Dans cette approche, le modèle reçoit en entrée, une instruction pour générer des tests unitaires avec JUnit 5 pour toutes les méthodes publiques d'une classe Java complète (`### Java class with all public methods`). Cela lui permet de travailler avec un contexte plus large et de prendre

en compte l'ensemble des méthodes de la classe lors de la création de tests, ce qui est particulièrement utile pour assurer une couverture maximale.
Dans ce prompt, nous avons donné au modèle des exemples de classes avec leurs tests unitaires correspondants pour lui permettre de "deviner" le format, les nuances et le type de logique qu'il doit appliquer pour générer une réponse qui correspond aux attentes.

En fournissant la classe complète, le modèle a accès à tous les détails nécessaires concernant les méthodes publiques, leurs signatures et les interactions possibles entre elles. Cela lui permettrait de concevoir des tests qui ne se contentent pas de vérifier le fonctionnement de chaque méthode individuellement, mais qui prennent également en compte les interactions entre elles et les scénarios d'utilisation réels.
En complément, le prompt inclut un ensemble de règles visant à orienter la génération et à garantir la production de tests exploitables. Ces règles n'ont pas été définies dès le départ, mais ajoutées progressivement au fil des expérimentations, en réponse aux erreurs de compilation ou aux problèmes de pertinence observés dans les tests générés initialement.
Les justifications de chaque règle dans *### Requirements* sont les suivantes :
Do not call private methods : respecter l'encapsulation et éviter des dépendances sur des comportements internes susceptibles de changer sans préavis.
Do not access private variables : prévenir toute manipulation directe de l'état interne de la classe, ce qui pourrait compromettre l'intégrité de l'objet testé.
Never use mock : limiter la complexité et éviter des scénarios artificiels qui ne reflètent pas l'exécution réelle du code.
Follow the AAA pattern : structurer chaque test en trois étapes (Arrange, Act, Assert) pour améliorer la clarté et la lisibilité des tests.
Cover critical use cases : garantir que les tests couvrent les cas standards, les cas limites et les scénarios d'erreur, afin d'augmenter la robustesse de la suite de tests.
Import all necessary functionalities : s'assurer que le code des tests est complet et autonome, évitant ainsi des erreurs liées à des dépendances manquantes.
Never instantiate an abstract class : respecter la nature abstraite de ces classes et éviter des comportements imprévisibles liés à leur instanciation.
Never test private methods : maintenir le focus sur le comportement observable de la classe plutôt que sur ses détails d'implémentation internes.

b. Few-shot prompting avec le contexte focal

```
### System Message:
As an assistant, I need to help a Java developer write a unit test for
the specified method of a given class using JUnit 5.

### Instruction:
Generate comprehensive JUnit 5 unit tests for the specified method of the given class,
ensuring maximum code coverage by thoroughly testing all possible scenarios and edge cases.

### Requirements:
- Do not call private methods: Private methods should never be directly invoked in unit tests.
- Do not access private variables: Private variables in the class should not be accessed or
manipulated directly in the tests.
- Never Use mock.
- Follow the AAA pattern: Structure each test according to the Arrange (set up the test
context), Act (execute the action being tested), and Assert (verify the result) pattern.
- Cover critical use cases: Ensure that the tests cover key scenarios, including normal cases,
edge cases, and error scenarios.
- Import all necessary functionalities: Make sure to import all classes and functions that are
required to perform the tests.
- Never instantiate an abstract class.

### Java Class with the specified method:
<example_focal_method_context>

### Unit tests
<example_tests>

### Instruction:
Generate comprehensive JUnit 5 unit tests for the specified method of the given class,
ensuring maximum code coverage by thoroughly testing all possible scenarios and edge cases.

### Requirements:
- Do not call private methods: Private methods should never be directly invoked in unit tests.
- Do not access private variables: Private variables in the class should not be accessed or
manipulated directly in the tests.
- Never Use mock.
- Follow the AAA pattern: Structure each test according to the Arrange (set up the test
context), Act (execute the action being tested), and Assert (verify the result) pattern.
- Cover critical use cases: Ensure that the tests cover key scenarios, including normal cases,
edge cases, and error scenarios.
- Import all necessary functionalities: Make sure to import all classes and functions that are
required to perform the tests.
- Never instantiate an abstract class.

### Java Class with the specified method:
<focal_method_context_to_test>

### Unit tests
```

Figure 11 : prompt du few-shot contexte focal

Dans cette approche, le modèle reçoit en entrée une instruction sur la création de tests unitaires avec JUnit 5 pour une méthode spécifique d'une classe Java (`### Java Class with the specified method`), ainsi que des exemples de tests déjà générés pour des méthodes similaires. L'objectif est de permettre au modèle de s'appuyer sur ces exemples (approche few-shot) pour guider la génération de nouveaux tests.

En fournissant des exemples de tests unitaires dans le prompt, le modèle peut mieux comprendre le format attendu et les types de scénarios à couvrir. Cela aide à réduire l'ambiguïté et à donner au modèle une référence concrète sur la manière de structurer ses propres tests. Les exemples servent de modèle, ce qui permet d'obtenir des tests plus ciblés et pertinents pour la méthode à tester.

3.2.4 Zero-Shot prompting

3.2.4.1 Généralités

a. Définition

Le *zero-shot prompting* [35] est une technique où un modèle de langage accomplit une tâche sans avoir été spécifiquement entraîné sur des exemples de cette tâche. Les modèles comme Mistral utilisent les vastes connaissances acquises lors de leur entraînement pour répondre à des requêtes ou accomplir des tâches en se basant uniquement sur les informations générales et les concepts qu'ils ont appris.

b. Structure Générale d'un Prompt

Pour créer un prompt efficace, il est important de structurer le message d'entrée de manière claire et concise. Voici les éléments clés à inclure dans la structure générale d'un prompt.

- **System Message (Facultatif mais Recommandé) :**

Le message à destination du modèle donne l'occasion de définir le contexte ou le comportement global attendu du modèle. Le message est utilisé pour guider le modèle sur la manière de répondre. Cela peut inclure le ton, le style ou le rôle que le modèle doit jouer. Par exemple : "Vous êtes un assistant utile qui fournit des informations claires et précises."

- **Instruction Spécifique (Prompt Principal) :**

L'instruction spécifique permet de décrire précisément la tâche ou la question à laquelle on souhaite que le modèle réponde. Elle doit être claire et concise pour maximiser la compréhension du modèle. Cette section est cruciale car elle détermine l'action que le modèle doit effectuer. Par exemple : "Écris un résumé de l'article suivant."

- **Contexte Additionnel (Facultatif) :**

Le contexte additionnel fournit des informations supplémentaires pour orienter la réponse du modèle. Il est utile pour des tâches nécessitant un contexte spécifique, comme des spécifications détaillées ou des exemples. Cela aide le modèle à comprendre mieux le contexte

ou les détails de la tâche. Par exemple : "L'article traite des impacts du changement climatique sur les récifs coralliens."

Ainsi, la clé pour un *zero-shot prompting* est la clarté et la précision des instructions. Un prompt bien défini aide le modèle à générer des réponses pertinentes et précises. Les prompts ambigus ou mal formulés peuvent entraîner des réponses incorrectes ou non pertinentes.

c. Avantages

Cette approche est flexible car le modèle peut répondre à une large gamme de questions et de tâches sans nécessiter un entraînement spécifique pour chaque tâche. Cela permet d'utiliser le même modèle pour diverses applications, que ce soit pour générer du code, répondre à des questions techniques, ou fournir des explications détaillées.

L'approche permet un gain de rapidité. En effet, il n'est pas nécessaire de créer et d'étiqueter des ensembles de données pour chaque nouvelle tâche. Le modèle peut immédiatement fournir des réponses ou des solutions basées sur les prompts donnés, ce qui réduit le temps et les ressources nécessaires pour préparer des données d'entraînement spécifiques.

Finalement, en termes d'adaptabilité, le modèle peut s'adapter à de nouvelles situations ou questions jamais vues auparavant. Cette capacité est particulièrement utile dans des environnements dynamiques où les exigences et les questions peuvent changer fréquemment.

d. Limitations

Les réponses peuvent être moins précises que celles obtenues avec des techniques nécessitant un entraînement spécifique. Le modèle peut parfois fournir des réponses qui ne sont pas totalement correctes ou qui manquent de détails.

Le modèle peut être moins performant pour des tâches très complexes ou nécessitant une compréhension approfondie de contextes spécifiques. Par exemple, générer du code pour des algorithmes complexes ou des systèmes intégrés peut nécessiter une compréhension plus fine que ce que le *zero-shot prompting* peut offrir, comme l'ont souligné Meredith Syed et Vrunda Gadesha (2025) dans leur analyse des limites de cette approche [36] .

3.2.4.2 Expérimentation avec Mistral

Dans notre recherche, nous avons adopté trois manières distinctes de faire du prompting avec Mistral 7B dans le but de générer des tests unitaires.

a. Zero-shot prompting avec le contexte global

```
### System Message:
As an assistant, I need to help a Java developer write a unit test for all public methods
of the given class using JUnit 5.

### Instruction:
Generate comprehensive JUnit 5 unit tests for all public methods of the given class,
ensuring maximum code coverage by thoroughly testing all possible scenarios and edge cases.

### Requirements:
- Do not call private methods: Private methods should never be directly invoked in unit
tests.
- Do not access private variables: Private variables in the class should not be accessed
or manipulated directly in the tests.
- Never Use mock.
- Follow the AAA pattern: Structure each test according to the Arrange (set up the test
context), Act (execute the action being tested), and Assert (verify the result) pattern.
- Cover critical use cases: Ensure that the tests cover key scenarios, including normal
cases, edge cases, and error scenarios.
- Import all necessary functionalities: Make sure to import all classes and functions that
are required to perform the tests.
- Never instantiate an abstract class.
- Never test private method.

### Java class with all public methods:
<class_to_test>

### Unit test:
```

Figure 12 : prompt du Zero-Shot Contexte Global

Dans cette approche, comme on peut le voir dans la *figure 12*, l'intégralité de la classe, comprenant tous ses champs, méthodes, et constructeurs, est fournie en entrée à Mistral. L'objectif est que le modèle génère des tests unitaires pour chaque méthode de la classe en tenant compte du contexte global de la classe.

Le modèle reçoit la classe complète, ce qui lui permettrait d'avoir une vue d'ensemble des interactions potentielles entre les méthodes, les dépendances, et les états partagés via les champs de la classe. Cette vue d'ensemble aiderait le modèle à produire des tests qui non

seulement vérifient les comportements individuels des méthodes, mais qui peuvent aussi capturer des scénarios où plusieurs méthodes interagissent entre elles.

- **System Message**

La section du "System Message" dans le prompt définit le contexte global et l'objectif de l'utilisateur. Ici, il est spécifié que l'utilisateur doit aider un développeur Java à écrire des tests unitaires pour toutes les méthodes publiques d'une classe donnée en utilisant JUnit 5. Ce message cadre la tâche en soulignant que tous les tests doivent être créés pour les méthodes publiques uniquement sans tester des méthodes privées. Ce cadre général prépare l'agent à la nature des tests à développer et aux outils qu'il doit utiliser, en insistant sur la création de tests exhaustifs.

- **Instruction**

La section "Instruction" détaille les exigences spécifiques que le test unitaire doit remplir. L'accent est mis sur la génération de tests pour toutes les méthodes publiques de la classe, en visant une couverture maximale du code. Il est demandé de suivre les meilleures pratiques en structurant les tests selon le modèle AAA (Arrange, Act, Assert) [37]. Chaque méthode publique doit être testée pour différents scénarios, incluant les cas normaux, les cas limites et les scénarios d'erreur. Ces derniers sont déduits par le modèle à partir des éléments disponibles dans le code source, comme les conditions de validation, les exceptions explicitement levées ou les comportements observables dans les implémentations. Cela garantit que les tests sont exhaustifs et qu'ils couvrent tous les cas d'utilisation possibles.

- **Requirements**

Dans cette section, les "Requirements" sont définis en termes de contraintes et d'importations nécessaires pour les tests. Les contraintes imposées incluent l'interdiction d'utiliser des méthodes privées, des variables privées etc. De plus, toutes les fonctionnalités nécessaires, y compris les bibliothèques JUnit et autres classes, doivent être correctement importées pour permettre l'exécution des tests sans erreur.

b. Zero-shot prompting avec le contexte focal

```
### System Message:
As an assistant, I need to help a Java developer write a unit test for the specified
method of a given class using JUnit 5.

### Instruction:
Generate comprehensive JUnit 5 unit tests for the specified method of the given class,
ensuring maximum code coverage by thoroughly testing all possible scenarios and edge
cases.

### Requirements:
- Do not call private methods: Private methods should never be directly invoked in unit
tests.
- Do not access private variables: Private variables in the class should not be accessed
or manipulated directly in the tests.
- Never Use mock.
- Follow the AAA pattern: Structure each test according to the Arrange (set up the test
context), Act (execute the action being tested), and Assert (verify the result) pattern.
- Cover critical use cases: Ensure that the tests cover key scenarios, including normal
cases, edge cases, and error scenarios.
- Import all necessary functionalities: Make sure to import all classes and functions
that are required to perform the tests.
- Never instantiate an abstract class.

### Java Class with the specified method:
```

Figure 13 : prompt Zero-Shot Contexte Focal

Dans cette approche, le modèle reçoit en entrée uniquement la méthode spécifique à tester, accompagnée des signatures des autres méthodes de la classe ainsi que des constructeurs. Le but est de permettre au modèle de se concentrer sur la génération de tests pour cette méthode particulière tout en disposant d'un contexte minimal pour comprendre les interactions possibles. En limitant les informations fournies au modèle à la seule méthode cible et à quelques éléments contextuels (comme les signatures des autres méthodes), cette approche vise à générer des tests plus ciblés et précis pour la méthode en question. Cela devrait réduire la complexité que le modèle doit gérer, ce qui peut conduire à des tests qui sont plus pertinents et adaptés aux comportements spécifiques de la méthode.

Le prompt demande au modèle de générer des tests unitaires en JUnit 5 pour cette méthode particulière, en se concentrant sur la couverture maximale pour cette méthode spécifique. Les mêmes directives sont données concernant l'accès aux méthodes privées, l'utilisation de mocks, et la structure des tests selon le schéma AAA [37].

En complément à ces deux premières méthodologies, nous avons exploré une troisième variante du zéro-shot prompting. Compte tenu de ses performances initiales, cette troisième approche a été retenue comme méthodologie finale. La section 3.3 détaille les principes et la mise en œuvre de cette approche adoptée, qui servira de base à nos analyses de résultats au chapitre suivant.

3.3 Méthodologie adoptée : Zero-Shot Prompting en Deux Temps (2TZSP)

Cette méthode décompose le processus de zero-shot prompting en deux interactions distinctes avec le modèle. Dans la première étape, on fournit une instruction générale et on recueille une réponse initiale du modèle. Dans la deuxième étape, on affine cette réponse en fournissant des instructions supplémentaires ou en clarifiant des aspects spécifiques. Cette approche vise à obtenir des résultats plus précis et personnalisés, car elle permet de corriger ou d'améliorer la réponse initiale du modèle.

- **Première phase : Identification des scenarios de test**

```
### Instructions:
As a Java developer, in the context of unit testing, list a maximum of 5 relevant scenarios to
test for the specified method of the following class:

        <focal_method_and_context>

        Desired output format start by "Test scenarios":
        Test scenarios:
        1. Test for <scenario_1>
        2. Test for <scenario_2>
        ...
        x. Test for <scenario_x>
```

Figure 14 : phase 1 Zero-Shot prompting en Deux Temps

Initialement, nous demandons à Mistral 7B de générer des scénarios de test possibles pour chaque méthode spécifiée. Bien que le modèle ne raisonne pas au sens strict, il est capable de produire des scénarios cohérents en s'appuyant sur les régularités apprises pendant son entraînement. En pratique, cela se traduit par une capacité à « simuler » des usages plausibles d'une méthode, à identifier des cas typiques d'entrée/sortie, ou encore à suggérer des cas limites (*figure 14*). Cette étape guide le modèle pour qu'il analyse la méthode en question et envisage diverses conditions d'entrée, chemins d'exécution et comportements de sortie qui pourraient être pertinents pour des tests exhaustifs.

Cette étape est cruciale car elle permet de décomposer le problème en sous-problèmes gérables et assure que tous les aspects significatifs de la méthode sont examinés. Le modèle utilise son entraînement et sa capacité à généraliser pour proposer une liste de cas de tests qui couvrent non seulement les fonctionnalités évidentes, mais aussi les cas limites et les exceptions potentielles.

- **Deuxième phase : Génération des Tests Unitaires**

```
### System Message:
As an assistant, I need to help a Java developer write a unit test for all public methods
of the given class using JUnit 5.

### Requirements:
- Do not call private methods: Private methods should never be directly invoked in unit
tests.
- Do not access private variables: Private variables in the class should not be accessed or
manipulated directly in the tests.
- Never Use mock.
- Follow the AAA pattern: Structure each test according to the Arrange (set up the test
context), Act (execute the action being tested), and Assert (verify the result) pattern.
- Cover critical use cases: Ensure that the tests cover key scenarios, including normal
cases, edge cases, and error scenarios.
- Import all necessary functionalities: Make sure to import all classes and functions that
are required to perform the tests.
- Never instantiate an abstract class.

### Instruction:
Generate unit test that covers all the following test scenarios for the specified method
of a given class using JUnit 5.
<generated_scenarios_first_step>

### Java Class with the specified method:
<focal_method_and_context>
```

Figure 15 : phase 2 Zero-Shot prompting en Deux Temps

Après avoir identifié les différents cas à tester, la seconde étape (*figure 15*) consiste à demander au modèle de générer les tests unitaires correspondants pour chaque cas identifié. À ce stade, le modèle est invité à produire des scripts de test concrets qui incluent non seulement les assertions nécessaires pour valider le comportement de la méthode sous test, mais aussi tout code de configuration requis.

Cet aspect du processus est essentiel pour transformer les spécifications théoriques ou les descriptions des cas de test en artefacts de tests exécutables qui peuvent être directement

utilisés dans un cadre de développement logiciel. Le modèle tire parti de sa compréhension des pratiques de codage, des conventions de nommage et des frameworks de test pour créer des tests qui sont non seulement fonctionnels mais aussi conformes aux standards de qualité du code.

3.4 Questions de recherche

Afin d'évaluer nos approches pour la génération de tests unitaires, nous avons formulé plusieurs questions de recherche. Ces questions visent à explorer divers aspects critiques de notre méthode, notamment sa capacité à générer des tests efficaces et pertinents, son efficacité par rapport à des outils établis, l'impact de différentes configurations et spécifications, ainsi que les implications économiques liées à l'utilisation d'une API payante pour ces tâches. Les réponses à ces questions fourniront un cadre d'évaluation complet, nous permettant de déterminer les forces et les limites de nos approches dans divers contextes.

3.4.1 Q1 Peut-on générer des tests unitaires pour les benchmarks de JUGE avec les différentes approches ?

Dans cette question de recherche, nous souhaitons évaluer les performances des différentes approches explorées sur des benchmarks utilisés lors de la compétition SBST 2020 [38] avec JUGE.
L'objectif de cette question de recherche est de voir les performances réelles de nos approches lorsqu'elles sont utilisées sur des systèmes volumineux et complexes, en particulier, si elles sont capables de générer des cas de test qui sont corrects, compilables et exécutables.

3.4.2 Q2 Comment ces approches se comparent-elles à EvoSuite ?

Cette question de recherche vise à comparer nos approches pour la génération de tests unitaires avec EvoSuite, un outil de référence dans le domaine.

3.4.3 Q3 Quelle est la qualité et la pertinence de ces tests unitaires ?

Cette question de recherche permet d'investiguer la qualité des tests unitaires générés par nos approches. Au-delà de leur capacité à compiler et à s'exécuter, il est crucial d'évaluer si ces tests sont pertinents (couvrent des chemins pertinents) et efficaces pour détecter des anomalies et assurer la qualité du code.

3.4.4 Q4 Quel est l'impact de la spécification des prompts ?

Cette question de recherche explore l'impact de la formulation des prompts sur la génération des tests unitaires à l'aide des grands modèles de langage. La manière dont les prompts sont rédigés peut influencer la qualité, la pertinence, et l'efficacité des tests produits. L'objectif est d'évaluer comment différentes spécifications de prompts affectent les résultats, et d'identifier les meilleures pratiques pour maximiser la performance des modèles de génération de tests unitaires.

3.4.5 Q5 Quel est l'impact du contexte spécifique ?

Cette question de recherche examine l'impact du contexte focal ou spécifique sur la génération des tests unitaires. L'objectif est de comprendre comment l'ajout d'informations contextuelles supplémentaires, telles que le nom de la classe, les constructeurs, les méthodes publiques, et les champs publics, influence la qualité, la pertinence, et la couverture des tests générés. Cette analyse permettra de déterminer si ce contexte enrichi améliore la capacité des modèles à produire des tests plus efficaces et adaptés aux méthodes testées.

Chapitre 4

Évaluation des modèles et discussion des résultats

Dans le cadre de l'évaluation des différentes approches étudiées pour la génération de tests unitaires basées sur des grands modèles de langage (LLM), nous avons sélectionné plusieurs projets issus de la compétition SBST 2020 [38]. Ces projets représentent des systèmes logiciels variés en termes de complexité, de domaine d'application et de structures de code, offrant un ensemble représentatif pour tester la robustesse et l'efficacité des tests générés. Voici un aperçu des projets utilisés dans notre étude comparative :

- **GUAVA** [39]: Une bibliothèque utilitaire Java développée par Google, fournissant des collections avancées, des caches, et de nombreux utilitaires pour la gestion des données. Grâce à ses structures de données sophistiquées, GUAVA présente des défis intéressants en matière de test, en particulier pour des scénarios d'interaction complexe entre objets.
- **SPOON** [40]: Un framework dédié à l'analyse et à la transformation de code Java, basé sur les Abstract Syntax Trees (AST). La manipulation de modèles de code abstraits en fait un cas complexe où la compréhension sémantique du code est cruciale pour la génération de tests pertinents.
- **PDFBOX** [41]: Une bibliothèque de manipulation de fichiers PDF, permettant la création, la modification et l'extraction de contenu. Ce projet implique des tests sur des données structurées complexes, notamment pour la gestion du texte, des images, et des métadonnées au sein de documents PDF.
- **FESCAR (Seata)** [42]: Un framework de gestion des transactions distribuées pour les microservices, permettant la coordination de multiples bases de données ou systèmes. La gestion des transactions distribuées constitue un défi de taille, en particulier pour tester les interactions et les cas d'échecs dans des environnements distribués.

Tableau 2 : Statistiques descriptives des projets de SBST 2020 [43]

Projet	# Classes	# Branches totales	Moyenne	Médiane	Ecart-Type
FESCAR	20	490	24,5	11	27,3
GUAVA	20	926	46,3	19	61,5
PDFBOX	20	1 070	53,5	24	75,6
SPOON	10	1 072	107,2	45	12,9

Le *tableau 2* montre que SPOON a le plus grand nombre de branches (1,072), bien qu'il ait seulement 10 classes, ce qui suggère une complexité élevée par classe. PDFBOX et GUAVA ont également un nombre élevé de branches, avec une moyenne de 46.3 et 53.5 branches par classe, respectivement. FESCAR a le nombre total de branches le plus bas (490), et une médiane très faible (11), ce qui peut indiquer que la complexité des branches est plus concentrée dans certaines classes.

Il faut noter que ces 70 classes (20 dans les projets GUAVA, FESCAR, PDFBOX et 10 dans SPOON) ont été retenues au terme d'une sélection. En effet, pour des raisons de temps et de faisabilité, toutes les classes des projets n'ont pas été incluses. Un processus de sélection a été appliqué afin de ne retenir qu'un sous-ensemble représentatif de classes présentant un niveau de complexité suffisant et compatibles avec les outils de génération de tests [38].

Les identifiants tels que GUAVA-39 ou PDFBOX-127 dans les *tableaux 4, 5*, et *6* correspondent à des identifiants uniques attribués aux classes du benchmark, combinant le nom du projet et un numéro arbitraire pour faciliter la traçabilité, la réplication des expériences et la référence précise aux classes analysées dans les différentes publications liées à la compétition SBST 2020 Tool.

4.1 RQ1 Peut-on générer des tests unitaires pour les benchmarks de JUGE avec les différentes approches ?

Afin de répondre à cette question, nous allons comparer les performances des 6 approches explorées. Leurs résultats sont présentés dans le tableau ci-dessous. Chaque approche a été évaluée sur le nombre total de tests générés (#TC), le nombre de tests réussis, c'est-à-dire les tests compilés et exécutés (PT) ; le nombre de tests brisés, c'est-à-dire les tests compilés mais qui échouent à l'exécution (BT) et le nombre de tests non compilables (UCT).

Tableau 3 : performances des approches

Approches	# TC	PT	BT	UCT	% PT	Observations
Zero-shot en deux temps	2490	533	363	1594	21,4%	La plus équilibrée entre le nombre total de tests générés et les tests réussis.
Zero-shot + contexte focal	2189	374	264	1551	17%	Légèrement en dessous de l'Approche 1 en termes de tests réussis.
Zero-shot + contexte global	1724	296	213	1215	17,2%	Taux d'échec relativement élevé.
Few-shot + contexte focal	1235	201	187	847	16,3%	Approche avec le plus faible ratio.
Few-shot + contexte global	561	288	38	235	51,3%	Approche générant peu de tests mais avec un bon taux de réussite.
Fine-tuning	231	62	39	130	26,8%	Très peu de tests générés, performances inferieures aux autres.

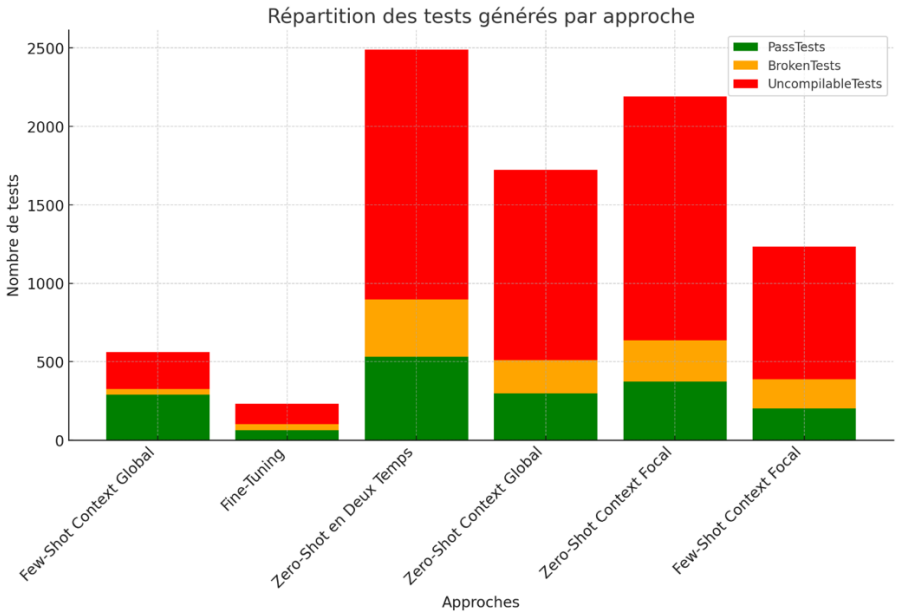

Figure 16 : Répartition des tests générés par approche

Le *tableau 3* et la *figure 16* nous montrent que toutes les approches explorées ont réussi à générer des portions de tests unitaires qui passent pour les benchmarks de JUGE. Cependant,

la quantité de tests générés et leur qualité (tests qui passent versus tests non compilables ou cassés) varient considérablement d'une approche à l'autre.

- **Nombre total de tests générés (#TC) :**

Les approches varient largement dans leur capacité à générer un volume élevé de tests unitaires. Certaines, comme l'approche 2TZSP (2490), ont généré plus de tests, même si la majorité des tests générés ne parviennent pas à passer la phase initiale de compilation. D'autres, comme le fine-tuning (231), ont généré un nombre beaucoup plus restreint.

Cela est lié à des différences dans la complexité dans le prompt et dans la qualité du jeu de données [32] (Method2test).

- **Nombre de tests réussis (PT) :**

Toutes les approches ont produit des tests qui passent, c'est-à-dire des tests compilables, exécutables et dont toutes les assertions sont satisfaites. Cela montre que les six approches explorées sont techniquement capables de générer des tests unitaires valides.

Toutefois, certaines approches, comme le 2TZSP, ont produit un nombre bien supérieur de tests réussis, même si le few-shot prompting avec le contexte global a obtenu un meilleur taux de réussite (51,3%), tandis que d'autres, comme le fine-tuning, ont généré beaucoup moins de tests exploitables.

- **Qualité des tests :**

Dans toutes les approches, une proportion importante (plus de la moitié) des tests générés étaient non compilables ou brisés, ce qui limite leur efficacité pratique malgré leur capacité à produire quelques tests valides.

- **Conclusion :**

Oui, il est possible de générer des tests unitaires pour les benchmarks de JUGE avec les 6 approches explorées. Toutes les approches ont réussi à produire un certain nombre de tests valides (PT), ce qui répond positivement à la question. Cependant, les performances des approches varient en termes de nombre total de tests générés et de proportion de tests exploitables.

4.2 RQ2 Comment ces approches se comparent-elles à EvoSuite ?

Nous avons comparé les performances de nos différentes approches à celles d'EvoSuite en termes de couverture (lignes et branches) et de score de mutation, avec un temps de génération de tests fixé à 180 secondes (par SBST 2020 [44]). Les résultats pour EvoSuite proviennent des métriques obtenues lors de la compétition SBST 2020 [44]. Nous allons présenter les

résultats des approches 2TZSP (meilleures performances), Few-Shot prompting Contexte Focal (performances moyennes) et le Fine-tuning (faibles performance). Afin de simplifier la présentation, nous désignerons notre outil par UtGen (Unit Test Generation) dans toutes les approches explorées.

4.2.1 Comparaison entre Zero-Shot prompting en Deux Temps vs Evosuite :

Tableau 4: "Zero-Shot prompting en Deux Temps" et Evosuite

Benchmark	Line Coverage		Branch Coverage		Mutation Score	
	utgen	evosuite	utgen	evosuite	utgen	evosuite
GUAVA-39	**46,7**	43,1	**30,2**	27,9	**41,1**	31,3
GUAVA-129	**37,1**	35,8	41,7	**42,5**	**50**	43,8
GUAVA-128	**63**	25	**73,7**	25,3	**68,3**	26,8
GUAVA-102	**71**	32,3	**63,2**	11,6	**77**	14,8
PDFBOX-127	31,2	**65,6**	7,1	**43,6**	16,7	**50,6**
PDFBOX-26	47,4	**56,8**	50	**100**	**100**	90
GUAVA-90	56,6	**89,6**	37,5	**90**	56,7	**89,3**
FESCAR-6	3,4	3,4	2,7	2,7	2,7	2,7
GUAVA-2	**80,2**	22,5	**54,7**	11,1	**77,1**	19,2
GUAVA-240	3,8	**22,7**	0	**5**	0	0
PDFBOX-91	78,9	**97,9**	37,5	**92,5**	**60**	0
PDFBOX-157	**20,6**	0	**4,3**	0	**5,5**	0
FESCAR-5	90,9	**99,1**	60	**98**	92,3	**100**
GUAVA-22	4,6	**49,7**	2,8	**47,3**	**6,9**	0
FESCAR-12	19,2	**100**	0	**87,5**	0	**100**
GUAVA-177	85,6	**98,5**	89,7	**99,3**	91,2	**100**
FESCAR-2	21,1	**25,8**	0	0	0	0
FESCAR-42	8,8	**42,6**	0	**27,1**	4,2	**25,4**
SPOON-169	4,1	**10,6**	0,8	**4,7**	0	**1,3**
PDFBOX-22	47,2	**63,9**	28,6	**64,3**	**38,1**	31,4
GUAVA-169	71,7	**86,7**	59,6	**85,3**	76,1	**89,3**
FESCAR-9	93,3	**98,3**	66,7	**100**	81,8	**98,2**
PDFBOX-285	67	**99,7**	27,3	**95,5**	**48,4**	0
PDFBOX-265	12	**70,2**	3,1	**52**	**5,9**	0
GUAVA-110	34,8	22,2	**25**	7,5	**25**	15
GUAVA-95	83,3	51,7	**50**	30	**75**	31,2
PDFBOX-234	83,7	**96,7**	66,7	**87,8**	**83,9**	20
SPOON-16	2,3	**16,1**	0	**9**	0	**6,4**
SPOON-253	17,5	**73,9**	6,2	**73,8**	**12,1**	0
FESCAR-37	39,4	**94,6**	26,5	**91,2**	**37,5**	0
FESCAR-41	1,7	1,7	2	2	2,4	2,4
PDFBOX-229	**80**	69,6	**100**	60	**92,9**	13,6
GUAVA-212	79,6	**94,3**	50	**83**	**68,4**	0
GUAVA-196	45	**70**	25	**75**	60	88
SPOON-155	9	**12,7**	0	**1,2**	**3,6**	3,2
SPOON-65	2,3	**9,7**	0	**8,9**	**0,6**	0
PDFBOX-198	35,4	**66,5**	7,3	**32,7**	**28,3**	0
GUAVA-47	**42,1**	11,3	**21,4**	0,7	**46,2**	0,4
PDFBOX-278	19,5	**98,3**	30	**96,3**	**19**	0
GUAVA-181	100	100	95,5	**100**	100	100
GUAVA-224	74,1	**89,6**	71,4	**90**	80,7	**90**
SPOON-20	9,5	**38,6**	0	**18**	0	**13,3**
Average	**43,44**	**56,13**	**31,39**	**49,53**	**41,32**	**30,89**

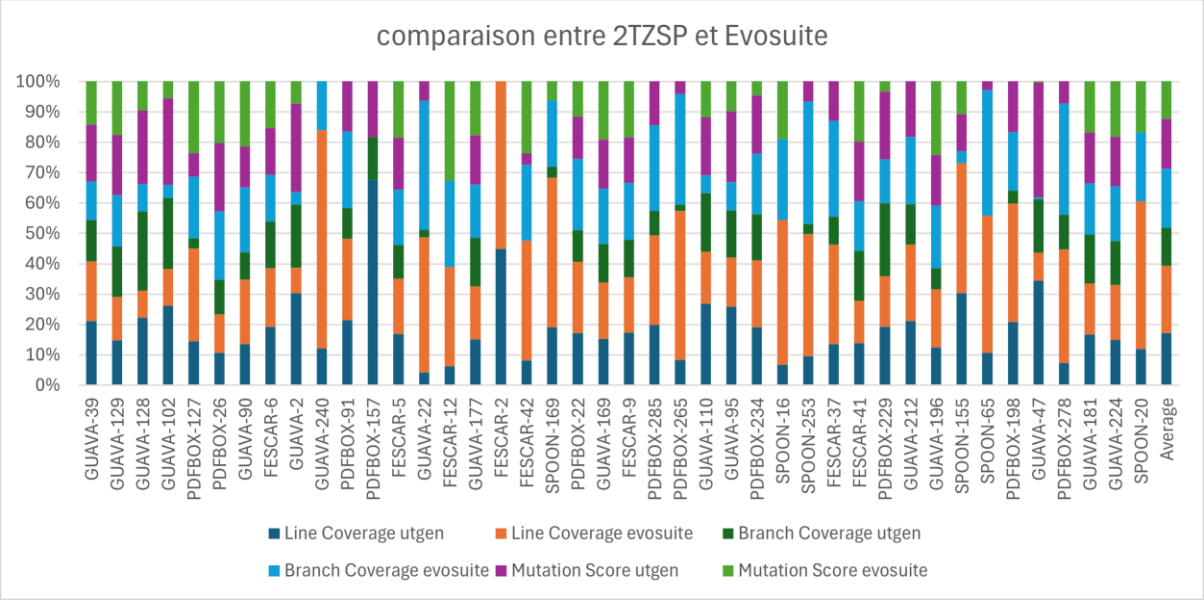

Figure 17 : comparaison entre le 2TZSP et Evosuite

- **Couverture de Lignes :**

Le *tableau 4* nous montre que sur les 42 classes testées par cette approche (2TZSP), EvoSuite a obtenu en moyenne, une couverture de lignes plus élevée avec un taux moyen de 56,13 % contre 43,44 % pour notre approche.

- **Couverture de Branches :**

EvoSuite montre également une meilleure performance dans la couverture de branches, avec un taux de 49,53 %, contre 31,39 % pour cette approche.

- **Score de Mutation :**

Sur cette métrique, notre approche 2TZSP surpasse EvoSuite avec des scores moyens respectifs de 41,32 % et 30,89 %.

Globalement, EvoSuite semble offrir une meilleure couverture de code, particulièrement en termes de couverture de lignes et de branches. Cependant, notre approche (2TZSP) montre des performances compétitives en termes de score de mutation. En effet, cette approche a un score de mutation supérieur à celui d'EvoSuite, ce qui suggère que ses tests sont plus efficaces pour détecter des erreurs dans le code.

4.2.2 Comparaison entre Few-Shot prompting Contexte Focal vs Evosuite

Tableau 5: "Few-shot prompting Contexte Focal" et Evosuite

Benchmark	Line Coverage		Branch Coverage		Mutation Score	
	utgen	evosuite	utgen	evosuite	utgen	evosuite
GUAVA-39	27,7	**43,1**	13,5	**27,9**	23,3	**31,3**
GUAVA-128	**51,1**	25	**50**	25,3	51,2	26,8
GUAVA-102	**55,6**	32,3	**39,5**	11,6	**60,7**	14,8
FESCAR-6	3,4	3,4	2,7	2,7	2,7	2,7
GUAVA-2	18,9	22,5	7,5	11,1	13,5	19,2
PDFBOX-91	42,1	97,9	25	92,5	20	0
FESCAR-5	50	**99,1**	10	**98**	46,2	**100**
GUAVA-22	17	49,7	13,9	47,3	24,8	0
GUAVA-177	58,7	98,5	56,9	99,3	57,5	100
FESCAR-2	21,1	25,8	0	0	0	0
PDFBOX-22	36,1	63,9	14,3	64,3	33,3	31,4
GUAVA-169	**8,7**	86,7	**4,4**	85,3	**9,2**	89,3
FESCAR-9	33,3	98,3	0	100	27,3	98,2
PDFBOX-285	34,1	**99,7**	**0**	**95,5**	18,8	**0**
PDFBOX-130	11,7	14,3	0	1,7	4	2,5
GUAVA-110	34,8	22,2	16,7	7,5	25	15
GUAVA-95	**75**	51,7	75	30	62,5	31,2
PDFBOX-234	74,4	96,7	55,6	87,8	77,4	20
SPOON-253	17,5	73,9	6,2	73,8	12,1	0
FESCAR-37	17	94,6	0	91,2	12,5	0
FESCAR-41	**1,7**	**1,7**	**2**	**2**	**2,4**	**2,4**
PDFBOX-229	**48**	69,6	**0**	60	**57,1**	13,6
GUAVA-212	76,8	94,3	25,7	83	50	0
GUAVA-196	40	70	50	75	60	88
SPOON-65	**2,1**	9,7	**0**	8,9	**0**	0
PDFBOX-198	11,8	66,5	0	32,7	9,8	0
SPOON-211	6,8	18,3	0	10,3	4,2	11,2
GUAVA-181	68,4	100	63,6	100	66,7	100
GUAVA-224	83,9	89,6	78,6	90	90	90
Average	**35,44**	**59,28**	**21,07**	**52,23**	**31,8**	**30,61**

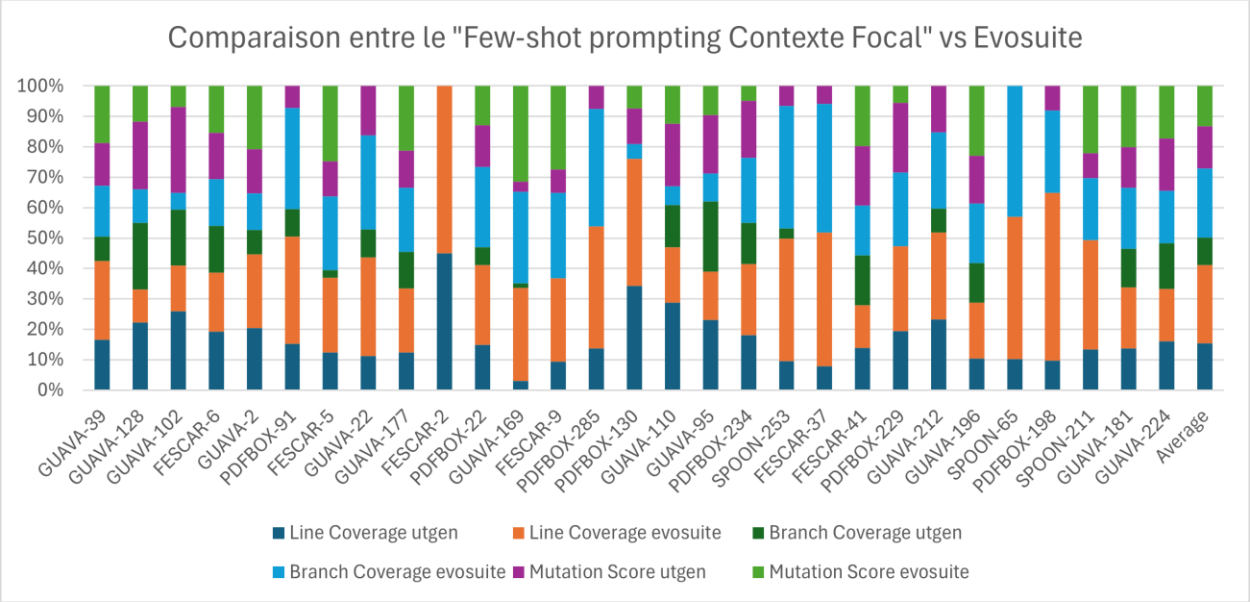

Figure 18 : comparaison entre le "Few-shot prompting Contexte Focal" et Evosuite

- **Couverture de Lignes :**

Comme on peut le voir dans le *tableau 5*, l'approche de Few-Shot prompting, en utilisant le contexte focal comme entrée, a atteint un taux moyen de couverture de lignes de 35,44 % sur les 29 classes testées, tandis qu'EvoSuite a obtenu une couverture de 59,28 %.

- **Couverture de Branches :**

Pour cette métrique, EvoSuite surpasse notre approche avec 52,23% de taux de couverture tandis que notre approche a obtenu 21,07%. EvoSuite couvre mieux les chemins de décision dans le code.

- **Score de Mutation :**

Malgré ces faibles performances en termes de couverture, en comparaison à celles d'EvoSuite, le « few-shot prompting avec le contexte focal » présente un score de mutation de 31,8%, plus élevé que celui d'EvoSuite (30,61%), écart d'environ 1 %. Bien que cet écart soit faible, il pourrait suggérer que, dans certains cas, cette approche parvient à détecter davantage de comportements incorrects. Toutefois, cette différence reste marginale et ne permet pas de conclure de manière définitive sur une supériorité globale en matière de détection de bogues.

4.2.3 Comparaison entre Fine-tuning vs Evosuite

Tableau 6: Fine-tuning et Evosuite

Benchmark	Line Coverage		Branch Coverage		Mutation Score	
	utgen	evosuite	utgen	evosuite	utgen	evosuite
GUAVA-39	**21,5**	43,1	**9,4**	27,9	**17,8**	31,3
GUAVA-128	58,7	25	55,3	25,3	70,7	26,8
GUAVA-102	55,6	32,3	57,9	11,6	59	14,8
PDFBOX-26	36,8	56,8	50	100	100	90
FESCAR-6	**3,4**	**3,4**	**2,7**	**2,7**	**2,7**	**2,7**
GUAVA-2	57,5	22,5	35,8	11,1	59,4	19,2
FESCAR-5	63,6	**99,1**	20	**98**	61,5	**100**
FESCAR-12	26,9	100	0	87,5	0	100
GUAVA-177	10,6	98,5	0	99,3	1,8	100
FESCAR-9	26,7	98,3	0	100	18,2	98,2
PDFBOX-285	46,2	99,7	13,6	95,5	34,4	0
PDFBOX-265	**14,8**	70,2	**4,7**	52	8,8	**0**
GUAVA-95	**41,7**	51,7	**0**	30	25	31,2
PDFBOX-234	14	**96,7**	**0**	**87,8**	9,7	**20**
SPOON-253	17,5	73,9	6,2	73,8	12,1	**0**
FESCAR-37	29,8	94,6	14,7	91,2	31,2	0
FESCAR-41	**1,7**	**1,7**	**2**	**2**	**2,4**	**2,4**
PDFBOX-40	24,4	54,9	10	46,6	13,4	0
PDFBOX-229	72	69,6	75	60	100	13,6
GUAVA-196	15	70	25	75	20	88
PDFBOX-198	47,2	66,5	14,6	32,7	39,1	0
GUAVA-181	18,4	**100**	18,2	**100**	21,4	**100**
GUAVA-224	7,1	89,6	5,7	90	8,6	90
Average	**30,92**	**66,00**	**18,30**	**60,87**	**31,18**	**40,36**

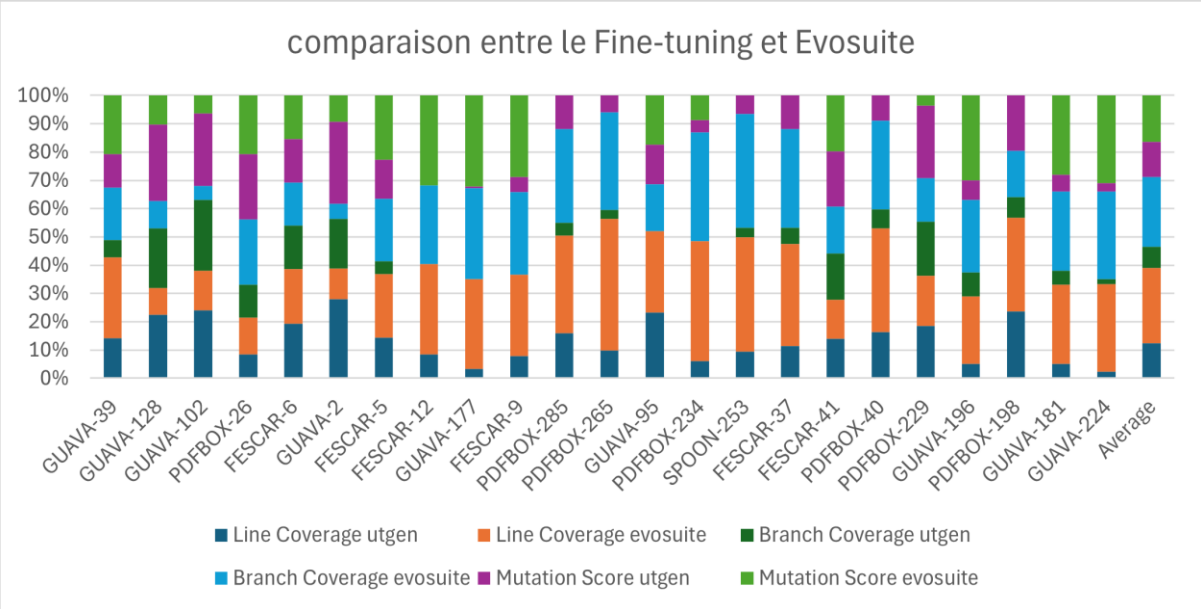

Figure 19 : comparaison entre l'approche Fine-tuning et Evosuite

- **Couverture de Lignes :**

Avec seulement 23 classes testées, l'approche fine-tuning a obtenu un faible taux de couverture de lignes (30,92%), comparée à celui d'EvoSuite qui est de 66%, comme on peut le voir dans le *tableau 6.*

- **Couverture de Branches :**

Pour cette métrique également, EvoSuite surpasse largement l'approche fine-tuning en obtenant un taux de 60,87% tandis que l'approche fine-tuning a eu un taux moyen de 18,30%.

- **Score de Mutation :**

Avec un score de mutation de 31,18 %, le modèle fine-tuné présente une efficacité inférieure à celle d'EvoSuite, dont le score atteint 40,36 %. Cela suggère qu'EvoSuite est davantage capable de détecter les défauts injectés dans le code.

4.2.4 Discussion des résultats

Pour cette analyse, nous nous concentrons sur l'approche qui a produit les meilleurs résultats globaux à savoir le 2TZSP. On pourrait se demander, pourquoi cette approche a été retenue malgré ses limites ?

- **Capacité à générer un grand nombre de tests**

C'est l'approche qui a produit le plus grand volume de cas de tests, ce qui est un avantage significatif. En effet, une plus grande diversité de tests signifie une meilleure exploration des comportements du code.

- **Nombre plus élevé de tests réussis**

Bien qu'elle ait généré un nombre élevé de **tests cassés et non compilables**, elle a aussi généré **plus de tests réussis** que les autres approches.

- **Meilleure couverture de code**

Malgré son ratio de succès plus faible, cette approche a atteint une meilleure couverture de lignes et de branches par rapport aux autres approches. Cela signifie qu'elle explore des parties du code que d'autres approches ne couvrent pas, même si certains tests générés sont incorrects.

- **Originalité et structuration en deux étapes**

Contrairement aux autres méthodes qui génèrent directement du code de test, **celle-ci décompose le processus en deux phases distinctes** :

  - **Phase 1 :** Génération de scénarios de test possibles pour les méthodes ciblées.
  - **Phase 2 :** Génération des cas de test correspondants pour chaque scénario identifié.

Cette approche structurée du 2TZSP permet de mieux organiser les tests en liant explicitement chaque test à un scénario identifié. Cette séparation rend la logique de génération plus claire, ce qui facilite la maintenance, l'ajout de nouveaux cas et l'amélioration progressive de la qualité des tests.

Bien que le 2TZSP ne présente pas le meilleur ratio de tests réussis, sa capacité à générer un grand volume de tests, sa couverture de code supérieure et son approche originale en deux étapes en font un choix pertinent. Nous allons maintenant examiner comment elle se compare à EvoSuite.

4.2.4.1 Couverture de Code et Algorithmes Génétiques d'EvoSuite

Les meilleures performances d'EvoSuite en termes de couverture de code sur la plupart des projets, hormis GUAVA, et celles de notre approche en score de mutation peuvent être expliquées par plusieurs facteurs liés aux caractéristiques de chaque outil et aux projets eux-mêmes :

a. Algorithme d'exploration exhaustive

EvoSuite utilise un algorithme génétique qui parcourt systématiquement les branches et les chemins du code pour maximiser la couverture de lignes et de branches. Il est ainsi bien adapté aux projets ayant un code linéaire ou des structures simples (comme SPOON, PDFBox et FESCAR), où les algorithmes génétiques peuvent générer rapidement une couverture large sans nécessiter une compréhension contextuelle approfondie du code.

b. Optimisation focalisée sur la couverture

Les algorithmes génétiques évoluent directement en fonction des métriques de couverture. EvoSuite peut donc atteindre des scores de couverture élevés sur des projets structurés où le flux de contrôle du code peut être exploré facilement et où la couverture de branches se traduit facilement par des tests supplémentaires.

4.2.4.2 Score de Mutation et Compréhension Contextuelle des LLM

a. Tests plus significatifs générés par les LLM

Bien que la couverture de code soit largement utilisée comme indicateur de la qualité des tests, plusieurs travaux ont montré qu'elle peut être trompeuse si utilisée seule. En effet, un test peut exécuter une ligne de code sans pour autant vérifier son comportement de manière effective. Ce phénomène est parfois qualifié de "vanity metric" [45], soulignant que la couverture peut donner une fausse impression de sécurité.

À l'inverse, le score de mutation permet d'évaluer la capacité réelle d'une suite de tests à détecter des erreurs. Cette métrique repose sur la génération de mutants (modifications mineures du code source), et sur l'observation de si les tests existants sont capables de les faire échouer. Si un test passe malgré un code modifié, cela révèle un manque d'assertions efficaces ou une lacune dans la conception des tests. Des études récentes confirment que le score de mutation offre une mesure plus fine de l'efficacité des tests que la simple couverture. Pour mesurer cet effet, Jain, K., et al. [46] introduisent la notion de "gap d'oracle", c'est-à-dire l'écart entre couverture et score de mutation, qui peut révéler des zones mal testées malgré une couverture élevée. De plus, Pedro Rijo [47] et BellSoft [48] soulignent que le test de mutation oblige les développeurs à écrire des assertions réellement utiles, rendant les tests plus robustes.

Ainsi, notre approche basée sur des LLM génère des tests susceptibles de prendre en compte le contexte et les interactions complexes entre les méthodes et les classes (les dépendances). Cela pourrait contribuer à détecter certains bogues plus subtils, mais cette hypothèse repose sur une interprétation des comportements observés dans certains tests générés et ne constitue pas une conclusion directement étayée par les métriques. Par ailleurs, le score de mutation obtenu par cette approche, supérieur à celui d'EvoSuite, suggère que ses tests peuvent, dans certains cas, cibler plus efficacement certaines défaillances dans le code.

b. Détection de scénarios d'échec plus profonds

Les LLM, en comprenant mieux les structures de données et les intentions des méthodes, créent des tests qui couvrent non seulement des chemins de code mais aussi des cas d'usage plus variés, ce qui permet de détecter des bogues cachés que les mutations révèlent.

4.2.4.3 Pourquoi GUAVA est l'exception ?

a. Structures de données complexes et algorithmes avancés

GUAVA contient des structures de données immuables, des caches, des collections complexes, et de nombreux algorithmes, qui nécessitent souvent des interactions fines et un haut niveau de contexte pour être testés efficacement. Ici, notre approche donne de meilleures performances qu'EvoSuite parce qu'elle peut interpréter et générer des tests plus pertinents pour ces interactions (prise en compte des interactions entre méthodes), alors qu'EvoSuite pourrait ne pas couvrir toutes les nuances de manipulation d'objets ou les algorithmes en profondeur (il trouve les séquences correctes uniquement par optimisation).

b. Limitations de l'algorithme d'EvoSuite dans les scénarios sophistiqués

Par « scénarios sophistiqués », nous faisons référence à des bibliothèques comme GUAVA, qui utilisent des structures de données avancées, une forte abstraction et des interactions internes complexes. Pour GUAVA, les algorithmes génétiques d'EvoSuite pourraient être moins efficaces dans certains cas, car l'outil vise principalement à maximiser la couverture sans nécessairement tenir compte des interactions complexes. Il est possible que les LLM génèrent, dans certains cas, des tests qui se concentrent davantage sur la logique et les fonctionnalités essentielles, ce qui pourrait être pertinent dans les bibliothèques utilitaires avancées comme GUAVA. Toutefois, cette observation repose sur une interprétation personnelle et ne constitue pas une conclusion directement étayée par les résultats obtenus. Bien que la couverture reste une métrique importante pour s'assurer qu'un maximum de code soit testé, le score de mutation donne une indication sur la capacité des tests à identifier des défauts. Cependant, un score de mutation élevé, pris isolément, ne garantit pas non plus la fiabilité des tests : obtenu avec une couverture faible, il ne reflète pas nécessairement une évaluation exhaustive du comportement du code. C'est pourquoi notre analyse combine plusieurs métriques afin d'offrir une évaluation plus équilibrée et représentative de la qualité des tests générés Ainsi, malgré des performances de couverture comparables pour GUAVA, notre approche a obtenu un score de mutation plus élevé, ce qui pourrait indiquer qu'elle constitue une option complémentaire ou potentiellement plus efficace pour certains aspects de la qualité du code.

4.2.4.4 Limites des autres approches

a. Le fine-tuning

L'approche de fine-tuning a montré des performances inférieures, avec seulement 30,92 % de couverture de lignes, 18,30 % de couverture de branches et 31,18 % de score de mutation. Ces résultats décevants peuvent être principalement liés à la qualité des données d'entraînement (Method2test [32]) et aux limitations inhérentes à l'approche.

- **Manque de contexte complet** : Les données utilisées pour le fine-tuning, issues de l'ensemble Methods2Test [32], bien que riches en cas de test et méthodes focales, manquaient de certains éléments essentiels pour simuler de manière réaliste l'environnement d'exécution des tests. Notamment, les extraits de code fournis étaient souvent dépourvus des importations de packages et des dépendances nécessaires, ce qui a pu limiter la capacité du modèle à comprendre et à reproduire le contexte complet dans lequel les méthodes testées fonctionnent.

En conséquence, les tests générés pouvaient manquer de pertinence pratique, échouant ainsi à détecter des bogues réalistes ou à couvrir des cas d'usage critiques (des situations clés telles que des erreurs de manipulation d'objet, des branches conditionnelles sensibles ou des exceptions qui ne surviennent que dans des configurations particulières).

- **Initialisation inadéquate des objets** : Dans les scénarios de test réels, l'initialisation appropriée des objets est cruciale pour évaluer de manière fiable le comportement des méthodes. Cependant, le modèle, en raison de lacunes dans les données d'entraînement, n'initialisait pas systématiquement les objets de manière correcte. Cela a conduit à la génération de tests potentiellement irréalistes (tests qui ne correspondent pas à des scénarios d'utilisation réels du logiciel) ou non exécutables, réduisant ainsi leur utilité pour une validation robuste du code.

b. Le few-shot prompting

L'approche few-shot prompting n'a pas non plus produit de résultats satisfaisants, et les performances obtenues pourraient être affectées par une surcharge dans le prompt.

- **Complexité excessive du prompt :** Avec cette approche, le prompt inclut non seulement des instructions spécifiques pour guider le modèle, mais aussi des exemples de tests ainsi que la classe ou la méthode à tester. Cette combinaison d'éléments complexes peut entraîner une surcharge cognitive pour le modèle, rendant plus difficile pour celui-ci, le traitement efficace de toutes les informations fournies. En conséquence, le modèle peut générer des tests qui ne répondent pas de manière optimale aux objectifs fixés, ce qui pourrait expliquer les faibles performances observées.
- **Perte de clarté des directives** : Lorsqu'un prompt est trop chargé, les instructions données au modèle peuvent être diluées ou perdre de leur clarté, ce qui complique la tâche du modèle pour produire des tests efficaces et pertinents. Le modèle peut ainsi avoir du mal à discerner les éléments les plus critiques du prompt, entraînant des résultats incohérents ou de faible qualité.

4.3 RQ3 Quelle est la qualité et la pertinence de ces tests unitaires ?

Pour répondre à cette question, nous nous concentrons sur l'approche qui a produit les meilleurs résultats globaux à savoir le 2TZSP.

L'évaluation de la qualité et de la pertinence des tests unitaires générés est essentielle pour comprendre leur efficacité dans un processus de développement logiciel. Cette analyse repose sur 3 critères clés, que nous pouvons appliquer aux tests générés avec cette approche.

- **Taux de Compilation :**

Parmi les tests générés, un pourcentage significatif (64 %) n'était pas compilable, ce qui indique que ces tests n'ont pas réussi à passer la première étape essentielle de la vérification syntaxique. Cela réduit automatiquement leur utilité.

- **Capacité de Détection d'Anomalies :**

Sur les tests qui se sont compilés et exécutés correctement, il est crucial de savoir s'ils détectent efficacement des anomalies dans le code. Les tests qui échouent à identifier des bogues existants sont moins pertinents, même s'ils s'exécutent sans erreur. Dans notre analyse, 21 % des tests générés sont corrects (compilés et exécutés avec succès) et ont contribué à une couverture de lignes de 43,44 % et une couverture de branches de 31,39 %. Cependant, la couverture de code ne suffit pas à elle seule à garantir la détection des anomalies.

- **Pertinence des Assertions :**

Le score de mutation obtenu par notre approche reflète la pertinence des assertions dans les tests générés, indiquant que ces tests parviennent à détecter 41,32 % des erreurs introduites intentionnellement dans le code. Bien que ce score montre que certaines mutations sont effectivement capturées, il révèle également que la majorité des erreurs passent inaperçues, suggérant que les assertions pourraient manquer de précision ou de portée.

En plus de ces critères, il est important de souligner que la lisibilité du code de test constitue un facteur essentiel pour garantir la maintenabilité, la compréhension et l'adoption des tests dans les projets logiciels. Plusieurs travaux ont montré que des tests difficiles à lire peuvent entraver la capacité des développeurs à identifier les intentions des tests, à détecter les erreurs logiques et à faire évoluer le système en toute confiance. Par exemple Sedano, T. [49] souligne que la lisibilité améliore la capacité des programmeurs à modifier et corriger le code tout en limitant les risques de régression. Ces constats sont renforcés par des recommandations industrielles, comme celles de Microsoft par Reese, J. [50] qui intègrent la lisibilité dans les bonnes pratiques de rédaction des tests unitaires. Enfin, l'étude plus récente de Delgado-Pérez, P., et al. [51] démontre que l'intégration d'une évaluation automatique de la lisibilité lors de la génération de tests favorise une meilleure acceptation des tests générés et une adoption accrue dans les projets logiciels. Dans notre étude, bien qu'aucune mesure quantitative de la lisibilité n'ait été réalisée, nous avons observé que les tests générés par notre approche respectent généralement les conventions de nommage et la structure recommandée par JUnit, contrairement à ceux produits par EvoSuite, où les noms de tests (ex. test0) révèlent davantage leur génération automatique. Cette observation reste qualitative et mériterait une évaluation plus systématique dans de futurs travaux.

4.4 RQ4 Quel est l'impact de la spécification des prompts ?

L'impact de la spécification des prompts est crucial, notamment lorsqu'il s'agit de la génération de tests unitaires à l'aide des grands modèles de langage (LLM). Les observations (différents types de formulations) montrent clairement que des prompts plus spécifiques et précis conduisent à des résultats significativement meilleurs.

Plusieurs travaux ont démontré que la qualité et la structure d'un prompt ont un impact significatif sur les performances des grands modèles de langage. Par exemple, l'approche Chain-of-Thought prompting [52] montre que la décomposition explicite du raisonnement dans le prompt améliore la capacité des modèles à résoudre des tâches complexes.

De plus, des études récentes sur l'ingénierie des prompts [53] [54] proposent des modèles de structuration efficaces qui influencent la qualité des réponses générées. Ces travaux confirment certaines tendances que nous avons observées dans nos expériences : l'approche 2TZSP, qui structure explicitement le prompt en étapes logiques, a obtenu, sur l'ensemble des projets évalués, de meilleures performances que toutes les autres approches explorées (few-shot prompting et zero-shot prompting direct), tant en couverture qu'en score de mutation. Cette supériorité suggère que la structuration et la contextualisation du prompt peuvent améliorer la pertinence et la qualité des tests générés.

4.4.1 Amélioration de la qualité des tests

- **Tests plus ciblés :** Des prompts détaillés permettent de générer des tests mieux alignés sur les comportements spécifiques à tester, augmentant ainsi la pertinence des tests pour les cas d'usage critiques.
- **Réduction des erreurs :** Les LLM ont parfois tendance à "halluciner" [55], c'est-à-dire à produire des informations incorrectes ou incohérentes lorsqu'ils reçoivent des instructions floues ou ambiguës. Des prompts précis aident à limiter ces erreurs en fournissant des directives claires et spécifiques, réduisant ainsi le nombre de tests non compilables ou erronés.

4.4.2 Réduction du bruit et des hallucinations

- **Moins de tests superflus et incorrects :** Lorsque les LLM sont confrontés à des prompts vagues, ils peuvent générer des tests superflus ou même incorrects en raison d'hallucinations. Des prompts bien définis limitent ces risques en cadrant strictement la génération, ce qui se traduit par des tests plus fiables et utiles.

- **Qualité sur quantité :** Un prompt précis favorise la génération de tests de haute qualité, en évitant le volume excessif de tests de faible valeur ou erronés que pourrait produire un prompt trop général.

4.5 RQ5 Quel est l'impact du contexte focal ?

Tableau 7: Contexte focal vs Contexte Global (classe complète)

Approch	Line Coverage	Branch coverage	Mutation Score
zero-shot + focal context	39,8	26,3	32,4
zero-shot + class	33,6	23,5	32,6
few-shot + focal context	35,4	21	31,8
few-shot + class	38,9	23,8	35,8

Le *tableau 7* résume les performances globales obtenues en termes de couverture de lignes, couverture de branches, et score de mutation pour différentes configurations d'approche et de contexte utilisés lors de la génération des tests unitaires.

- **Couverture de Lignes :**

L'approche "zero-shot + focal context" montre une couverture de ligne supérieure à celle du "zero-shot + class", avec 39,8 % contre 33,6 %, indiquant que le contexte focal aide à explorer davantage de lignes de code.

En mode "few-shot", l'effet du contexte focal reste bénéfique : "few-shot + focal context" atteint une couverture de lignes de 35,4 %, inférieure à celle du "few-shot + class" qui est de 38,9 %. La couverture moyenne, bien que légèrement meilleure dans l'approche "few-shot + class", peut ne pas toujours être représentative de l'impact global. En effet, le "few-shot + focal context" teste un plus grand nombre de classes (29 classes) que le "few-shot + class" (15 classes). Cela suggère que le contexte spécifique pourrait être plus efficace pour une exploration large du code, garantissant que davantage de parties du système soient testées, bien que chaque classe ne soit peut-être pas couverte en profondeur.

- **Couverture de Branches :**

La couverture de branches suit une tendance similaire. "zero-shot + focal context" obtient une couverture de branches de 26,3 %, contre 23,5 % pour "zero-shot + class". En mode "few-shot", l'approche avec le contexte spécifique atteint 21 %, tandis que la classe complète obtient 23,8 %. Cette différence de 2,8% ne signifie pas que l'approche few-shot avec la classe complète est globalement plus efficace pour couvrir divers chemins d'exécution. En effet, il faut tenir compte du fait que le "few-shot + focal context" permet de tester plus de classes.

- **Score de Mutation :**

L’approche avec la classe complète (ou le contexte global) donne un meilleur score de mutation en mode "zero-shot" comme en "few-shot" même si la différence est légère surtout avec le "zero-shot".

En "zero-shot", le contexte focal surpasse le contexte global en termes de couverture de lignes et de branches, ce qui indique qu’il permet une meilleure exploration initiale des chemins d'exécution. En mode "few-shot", toutefois, le contexte global reprend légèrement l'avantage, bien que cette différence de couverture moyenne ne rende pas forcément l'approche plus efficace dans un contexte global.

Pour des projets nécessitant une vérification d'une large étendue fonctionnelle sans se concentrer excessivement sur les détails de chaque classe, le contexte focal pourrait donc être plus pertinent. En revanche, pour des systèmes nécessitant une validation approfondie de certains modules clés, l'approche avec classe complète est probablement plus bénéfique.

Toutefois, il faut savoir qu’en donnant au modèle la classe au complet, l'approche de contexte global peut entraîner une surcharge d'informations, diluant l'attention du modèle et diminuant la qualité des tests générés. Le modèle pourrait avoir du mal à discerner les éléments les plus critiques parmi un grand nombre de méthodes et de détails non pertinents.

Chapitre 5

Conclusion et perspectives

L'objectif de cette étude était d'explorer comment automatiser la génération de tests unitaires en utilisant les grands modèles de langage (LLM), tout en les comparant à un outil bien établi, EvoSuite. Les tests unitaires sont cruciaux pour assurer la qualité du code et faciliter sa maintenance. Cependant, leur génération manuelle est une tâche coûteuse en temps et sujette à l'erreur. Nous avons ainsi cherché à répondre à la question suivante : *Peut-on générer des tests unitaires efficaces en utilisant des approches basées sur les LLM, et si oui, lesquelles sont les plus performantes* ?

À travers des expériences et des comparaisons utilisant différentes métriques (couverture de lignes, couverture de branches, score de mutation), nous avons pu constater que, bien que les LLM puissent effectivement générer des tests unitaires, toutes les approches testées n'ont pas produit des résultats satisfaisants. Les stratégies basées sur le fine-tuning ont montré des limites importantes liées à la qualité des données d'entraînement et au manque de contexte nécessaire pour générer des tests réalistes et fonctionnels. En revanche, les approches de prompting, notamment celles basées sur un contexte focal, ont montré une meilleure capacité à générer des tests pertinents, tout en restant compréhensibles.

Malgré ces avancées, plusieurs défis subsistent. Les tests générés par les LLM peuvent encore souffrir de problèmes de compilation ou de pertinence, et la couverture du code générée reste inférieure à celle des outils spécialisés comme EvoSuite. De plus, l'interprétation des prompts et la gestion du contexte dans les grands modèles de langage posent des défis qui nécessitent des ajustements fins pour obtenir des résultats performants.

Les limites de cette recherche tiennent d'une part, à la qualité des données d'entraînement, qui peut influencer la pertinence des résultats obtenus, et d'autre part, à la complexité inhérente liée à la conception et à l'optimisation de prompts efficaces, un processus délicat lorsqu'il s'agit des LLM dont le comportement peut varier considérablement selon la formulation et la structure des instructions. Par ailleurs, la dépendance à des données spécifiques et la nécessité de combiner des approches restent des obstacles pour une automatisation complète et fiable de la génération de tests.

Dans la continuité de cette étude, plusieurs pistes de recherche méritent d'être explorées.

- **Fournir un contexte plus riche au modèle :** Pour la première phase de l'approche 2TZSP**,** on pourrait ajouter des descriptions plus précises du comportement attendu des méthodes (préciser les types d'entrées et sorties valides pour éviter les scénarios absurdes).
- **Intégrer une étape de filtrage automatique des scénarios :** On pourrait utiliser des règles heuristiques ou un post-traitement pour éliminer les scénarios peu pertinents ou redondants dans la première phase. On pourrait aussi mettre en place une validation syntaxique et logique des scénarios avant la phase 2 du 2TZSP.
- Pendant la phase 2, on pourrait **automatiser la correction des erreurs de compilation** en ajoutant un post-traitement des tests générés. Celui-ci consisterait à détecter les erreurs syntaxiques à l'aide d'un compilateur ou d'un analyseur statique (outils qui analysent le code sans l'exécuter, pour détecter des erreurs de compilation, des incohérences ou des vulnérabilités), puis à appliquer un module de correction automatique pour ajuster les erreurs avant l'exécution.
- Améliorer la qualité et la diversité des données d'entraînement pour le fine-tuning pourrait offrir des gains significatifs en termes de pertinence des tests générés.
- Il serait intéressant aussi d'explorer des approches hybrides combinant les points forts des méthodes de fine-tuning et de prompting pour maximiser à la fois la couverture et la compréhension du contexte du code. Enfin, l'intégration de feedback en boucle pourrait permettre d'affiner les tests générés en fonction des résultats de leur exécution, améliorant ainsi leur qualité de manière itérative.

En conclusion, bien que des progrès notables aient été réalisés avec les LLM dans la génération de tests unitaires, il reste encore des défis à relever avant de pouvoir automatiser pleinement cette tâche de manière fiable et généralisée. Toutefois, les résultats prometteurs de certaines approches (2TZSP) suggèrent que les LLM ont un potentiel clé à jouer dans l'automatisation des tests.

Bibliographie

[1] Achiam, J., et al., *Gpt-4 technical report.* arXiv preprint arXiv:2303.08774, 2023.

[2] Devlin, J., *Bert: Pre-training of deep bidirectional transformers for language understanding.* arXiv preprint arXiv:1810.04805, 2018.

[3] Colin, R., *Exploring the limits of transfer learning with a unified text-to-text transformer.* J. Mach. Learn. Res., 2020. **21**: p. 140: 1–140: 67.

[4] Vaswani, A., *Attention is all you need.* Advances in Neural Information Processing Systems, 2017.

[5] Fraser, G. and A. Arcuri, *A large-scale evaluation of automated unit test generation using evosuite.* ACM Transactions on Software Engineering and Methodology (TOSEM), 2014. **24**(2): p. 1-42.

[6] Fraser, G. and A. Arcuri. *Evolutionary generation of whole test suites*. in *2011 11th International Conference on Quality Software*. 2011. IEEE.

[7] *junit-quickcheck*. Available from: https://github.com/pholser/junit-quickcheck.

[8] Fraser, G. and A. Arcuri. *Evosuite: automatic test suite generation for object-oriented software*. in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 2011.

[9] Panichella, A., F.M. Kifetew, and P. Tonella, *Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets.* IEEE Transactions on Software Engineering, 2017. **44**(2): p. 122-158.

[10] Panichella, A., F.M. Kifetew, and P. Tonella. *Reformulating branch coverage as a many-objective optimization problem*. in *2015 IEEE 8th international conference on software testing, verification and validation (ICST)*. 2015. IEEE.

[11] Pacheco, C. and M.D. Ernst. *Randoop: feedback-directed random testing for Java*. in *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. 2007.

[12] Albert, E., et al. *jpet: An automatic test-case generator for java*. in *2011 18th Working Conference on Reverse Engineering*. 2011. IEEE.

[13] Tillmann, N. and J. De Halleux. *Pex–white box test generation for. net*. in *International conference on tests and proofs*. 2008. Springer.

[14] El‐Far, I.K. and J.A. Whittaker, *Model‐based software testing.* Encyclopedia of software engineering, 2002.

[15] Tufano, M., et al., *Unit test case generation with transformers and focal context.* arXiv preprint arXiv:2009.05617, 2020.

[16] Lewis, M., *Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension.* arXiv preprint arXiv:1910.13461, 2019.

[17] Chen, M., et al., *Evaluating large language models trained on code.* arXiv preprint arXiv:2107.03374, 2021.

[18] Yuan, Z., et al., *Evaluating and improving chatgpt for unit test generation.* Proceedings of the ACM on Software Engineering, 2024. **1**(FSE): p. 1703-1726.

[19] Sapozhnikov, A., et al. *TestSpark: IntelliJ IDEA's Ultimate Test Generation Companion*. in *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*. 2024.

[20] Jiang, A.Q., et al., *Mistral 7B.* arXiv preprint arXiv:2310.06825, 2023.

[21] Dettmers, T., et al., *Qlora: Efficient finetuning of quantized llms.* Advances in neural information processing systems, 2023. **36**: p. 10088-10115.

[22] Lutkevich, B. *Hugging Face*. 2023; Available from: https://www.techtarget.com/whatis/definition/Hugging-Face#:~:text=Hugging%20Face%20lets%20users%20create,and%20test%20models%20more%20easily.
[23] Devroey, X., et al., *JUGE: An infrastructure for benchmarking Java unit test generators.* Software Testing, Verification and Reliability, 2023. **33**(3): p. e1838.
[24] *SBST 2020*. 2020; Available from: https://sbst20.github.io/.
[25] Al-Ahmad, B., et al., *Jacoco-coverage based statistical approach for ranking and selecting key classes in object-oriented software.* Journal of Engineering Science and Technology, 2021. **16**(08): p. 2021.
[26] Coles, H., et al. *Pit: a practical mutation testing tool for java*. in *Proceedings of the 25th international symposium on software testing and analysis*. 2016.
[27] Alfsson, O., *An analysis of Mutation testing and Code coverage during progress of projects*. 2017.
[28] Petrović, G., et al. *Does mutation testing improve testing practices?* in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 2021. IEEE.
[29] Tomassetti, F., et al., *JavaParser*. 2021.
[30] Alon, U., et al., *code2vec: Learning distributed representations of code.* Proceedings of the ACM on Programming Languages, 2019. **3**(POPL): p. 1-29.
[31] Kassel, R., *Fine-Tuning : Qu'est-ce que c'est ? À quoi ça sert en IA ?* 2024.
[32] Tufano, M., et al. *Methods2Test: A dataset of focal methods mapped to test cases*. in *Proceedings of the 19th International Conference on Mining Software Repositories*. 2022.
[33] Hu, E.J., et al., *Lora: Low-rank adaptation of large language models.* arXiv preprint arXiv:2106.09685, 2021.
[34] DAIR.AI. *Few-Shot Prompting*. 2025; Available from: https://www.promptingguide.ai/techniques/fewshot.
[35] DAIR.AI. 2025; Available from: https://www.promptingguide.ai/techniques/zeroshot.
[36] Gadesha, M.S.e.V., *What is zero-shot prompting?* 2025.
[37] Sathi, C. *Arrange Act Assert Pattern in Unit Testing*. 2024; Available from: https://codersathi.com/arrange-act-assert-pattern/.
[38] Devroey, X., S. Panichella, and A. Gambi. *Java unit testing tool competition: Eighth round*. in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*. 2020.
[39] Google, *Guava*.
[40] Pawlak, R., et al., *Spoon: A library for implementing analyses and transformations of java source code.* Software: Practice and Experience, 2016. **46**(9): p. 1155-1179.
[41] Apache, *PDFBox*. 2020.
[42] Kailing, C. *Unveiling the Principles of Fescar Distributed Transaction*. 2019; Available from: https://seata.apache.org/blog/seata-analysis-simple/.
[43] Herlim, R.S., et al. *Empirical study of effectiveness of evosuite on the sbst 2020 tool competition benchmark*. in *Search-Based Software Engineering: 13th International Symposium, SSBSE 2021, Bari, Italy, October 11–12, 2021, Proceedings 13*. 2021. Springer.
[44] Panichella, A., J. Campos, and G. Fraser. *Evosuite at the sbst 2020 tool competition*. in *Proceedings of the IEEE/ACM 42nd international conference on software engineering workshops*. 2020.
[45] Codecov, *Mutation Testing: How to Ensure Code Coverage Isn't a Vanity Metric*. 2022.

[46] Jain, K., et al. *Mind the gap: The difference between coverage and mutation score can guide testing efforts*. in *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. 2023. IEEE.
[47] Rijo, P., *An intro to Mutation Testing - or why coverage sucks*. 2019.
[48] Bellsoft, *A Comprehensive Guide to Mutation Testing in Java*. 2025.
[49] Sedano, T. *Code readability testing, an empirical study*. in *2016 IEEE 29th International conference on software engineering education and training (CSEET)*. 2016. IEEE.
[50] Reese, J., *Unit testing best practices for .NET.* 2025.
[51] Delgado-Pérez, P., et al., *InterEvo-TR: Interactive evolutionary test generation with readability assessment.* IEEE Transactions on Software Engineering, 2022. **49**(4): p. 2580-2596.
[52] Wei, J., et al., *Chain-of-thought prompting elicits reasoning in large language models.* Advances in neural information processing systems, 2022. **35**: p. 24824-24837.
[53] Sahoo, P., et al., *A systematic survey of prompt engineering in large language models: Techniques and applications.* arXiv preprint arXiv:2402.07927, 2024.
[54] White, J., et al., *A prompt pattern catalog to enhance prompt engineering with chatgpt.* arXiv preprint arXiv:2302.11382, 2023.
[55] Xu, Z., S. Jain, and M. Kankanhalli, *Hallucination is inevitable: An innate limitation of large language models.* arXiv preprint arXiv:2401.11817, 2024.