Actes de CORIA-TALN-RJCRI-RECITAL 2025
Marseille, France, 30 juin au 4 juillet 2025

Amélioration et Automatisation de la Génération des Cas de Tests Logiciels à l'Aide du Modèle Llama

MOUGHIT Imane[1,] HAFIDI Imad [2]

(1) Laboratoire LIPIM, École Nationale des Sciences Appliquées, Université Sultan Moulay Slimane, Khouribga, Maroc
(2) Laboratoire LIPIM, École Nationale des Sciences Appliquées, Université Sultan Moulay Slimane, Khouribga, Maroc
Moughit.iman@gmail.com, i.hafidi@usms.ma

RESUME

L'émergence des Large Language Models (LLM) a révolutionné l'ingénierie logicielle grâce à leurs capacités de compréhension et de génération du langage naturel. Bien qu'ils soient utilisés pour la génération automatique de cas de test, les approches actuelles reposant uniquement sur les méthodes focales ou sur des descriptions textuelles présentent des limites: elles peinent à capturer les comportements attendus, les cas limites et les scénarios d'erreur, et sont peu compatibles avec le développement piloté par les tests (TDD). Pour répondre à ces contraintes, nous proposons une approche hybride (Texte, Méthodes focales → Cas de test), combinant les commentaires présents dans le code avec la logique de la méthode cible. En exploitant le modèle LLaMA 3-8B et des techniques de prompt engineering, ainsi que l'évaluation des cas de test générés à l'aide d'un LLM en tant que juge, notre méthode vise à automatiser et améliorer la génération des cas de test. Testée sur des projets open source, elle a permis de générer 7 606 cas de test, avec un taux de correction syntaxique de 97 %.

ABSTRACT

Improvement and Automation of Software Test Case Generation Using the Llama Model

The emergence of Large Language Models (LLM) has revolutionized software engineering through their advanced capabilities in natural language understanding and generation. Although these models are increasingly used for automated test case generation, current approaches that rely solely on focal methods or textual descriptions face significant limitations: they struggle to capture expected behaviors, edge cases, and error scenarios, and are poorly aligned with Test-Driven Development (TDD) practices. To address these challenges, we propose a hybrid approach (Text, Focal Methods → Test Cases) that combines code comments with the logic of the target method. By leveraging the LLaMA 3-8B model and prompt engineering techniques, along with the evaluation of generated test cases using an LLM as a judge, our method aims to automate and improve test case generation. Tested on open-source projects, it successfully generated 7,606 test cases, with a syntax correction rate of 97%.

MOTS-CLES : LLaMA 3-8B, génération cas de test, ingénierie logiciel, TDD, LLM en tant que juge.

KEYWORDS : LLaMA 3-8B, test case generation, software engineering, TDD, LLM as a judge.

Actes de l'atelier Évaluation des modèles génératifs (LLM) et challenge 2025 (EvalLLM), pages 22–35.

1 Introduction

À l'heure actuelle, le développement logiciel s'impose comme une nécessité incontournable. L'évolution constante de la taille et de la complexité des applications rend les méthodes de test traditionnelles de moins en moins adaptées. Les tests reposant sur l'intelligence artificielle (IA) s'imposent aujourd'hui comme une nouvelle référence, apportant des améliorations notables en matière de qualité et de fiabilité des logiciels. Avec des systèmes logiciels de plus en plus complexes, les méthodes de test doivent évoluer pour faire face à cette complexité. Parallèlement, l'accélération des cycles de développement, notamment avec les méthodologies agiles et la livraison continue, rend indispensable l'automatisation des tests par l'IA afin de maintenir des cadences de publication rapides sans compromettre la qualité.

Alors que les utilisateurs exigent des produits sans défaut, les tests basés sur l'IA permettent d'atteindre un niveau de qualité plus élevé, répondant efficacement à ces attentes accrues. En optimisant les ressources, les tests automatisés par l'IA réduisent le temps consacré aux tests manuels, libérant ainsi du temps pour d'autres tâches essentielles et diminuant les coûts opérationnels. De plus, les tests IA offrent une évolutivité exceptionnelle, s'adaptant facilement à des applications plus vastes ou plus complexes sans nécessiter une augmentation proportionnelle des efforts ou des coûts.

Ce qui distingue véritablement les tests basés sur l'IA des méthodes traditionnelles, c'est leur capacité à exploiter l'apprentissage automatique pour optimiser les processus de test. Contrairement aux approches classiques, où les cas de test sont créés manuellement, l'IA apprend des modèles présents dans les données de test et s'ajuste dynamiquement à l'évolution du logiciel. Cela garantit une gestion continue et complète de la qualité, faisant de l'IA un outil incontournable dans le monde du développement logiciel rapide et dynamique.

Par ailleurs, l'émergence des grands modèles de langage (Large Language Models, ou LLM) a révolutionné l'ingénierie logicielle. Ces modèles ont réalisé des avancées significatives, démontrant des capacités proches de l'intelligence humaine. Au cours des dernières années, les LLM ont été largement adoptés dans diverses tâches d'ingénierie logicielle. Comme le soulignent les articles de revue (Wang, 2024), (Hou, 2023), (Fan, 2023), les LLM ont montré des performances prometteuses dans de nombreuses tâches de développement et de maintenance logicielle, telles que la génération de programmes (Dong, 2023), (Yetiştiren, 2023),(Li, 2023), (Liu, 2024), (Liu, 2023) , les tests logiciels (Deng, 2023), (Endres, 2024) et l'amélioration de programmes (Gong, 2025), (Poldrack, 2023) et (Cummins, 2024). L'intégration des LLM dans les tests basés sur l'IA renforce encore davantage la capacité à gérer la complexité croissante et les cycles de développement rapides des systèmes logiciels modernes, en garantissant des processus de test efficaces, adaptables et fiables.

Dans cet article, nous nous concentrerons sur la génération de cas de test à l'aide du modèle LLaMA 3-8B. Pour ce faire on a crée une base de données à partir de plusieurs projets open source.

2 Contexte et Travaux connexes

Dans cette section, nous introduisons les concepts fondamentaux du test logiciel, des cas de test et des Large Language Models (LLM), avant de passer en revue les travaux connexes pour justifier et motiver notre étude.

2.1 Test logiciel et conception des cas de test

Le test logiciel est un processus constitué de toutes les activités du cycle de vie, statiques et dynamiques, concernant la planification, la préparation et l'évaluation d'un produit logiciel et des artefacts associés, afin de vérifier qu'ils satisfont aux exigences spécifiées, de démontrer qu'ils sont adaptés à l'usage, et d'identifier les éventuels défauts tel que indiqué dans le syllabus ISTQB Certified Tester Foundation Level Syllabus v4.0.

Une idée reçue fréquente consiste à penser que tester un logiciel se limite à exécuter des tests, c'est-à-dire à faire fonctionner l'application et à vérifier ses résultats. En réalité, le test logiciel comprend également des activités complémentaires, telles que la revue de documents, la conception des cas de test, la mise en place des environnements de test, ou encore l'analyse des anomalies. Il s'agit d'un processus complet, qui doit être intégré à chaque étape du cycle de développement logiciel.

Parmi les différentes activités de test, la **conception des cas de test** qui occupe une place centrale. Un **cas de test** se définit comme : « Un ensemble de conditions préalables, d'entrées, d'actions (le cas échéant), de résultats attendus et de conditions postérieures, développé sur la base d'éléments de test, pour un objectif de test particulier, tel que l'exercice d'un chemin particulier du programme ou la vérification de la conformité à une exigence spécifique. »

Les cas de test jouent un rôle central dans l'activité de test logiciel. Ils permettent non seulement de détecter d'éventuels bugs, mais aussi de vérifier que les fonctionnalités testées répondent bien aux attentes. Grâce à eux, on peut s'assurer qu'une fonctionnalité a été évaluée de manière complète et cohérente. Pour être efficaces, les cas de test doivent suivre certaines règles de structuration afin d'être les plus clairs, complets et exhaustifs possible. Le critère essentiel d'un bon cas de test est sa lisibilité : il doit pouvoir être compris et exécuté par n'importe qui, même sans connaissance préalable du contexte.

Un cas de test bien rédigé doit :

- Avoir un titre explicite et significatif ;
- Suivre une structure claire : numéro du cas > étape (step) > résultat attendu. Chaque étape doit commencer par un verbe d'action (par exemple : cliquer, naviguer, ouvrir, scroller...) ;
- Indiquer les prérequis si nécessaire, comme un jeu de données ou un paramétrage spécifique à effectuer au préalable ;
- Ne pas dépasser 15 étapes, sous peine d'être trop complexe : dans ce cas, il convient de le découper davantage ;
- Être conçu du point de vue de l'utilisateur, pour garantir la pertinence des scénarios testés.

En structurant ainsi les tests, les cas de test assurent la formalisation du processus de validation. Ils précisent ce qui doit être testé, dans quelles conditions, et quels résultats sont attendus. Ils contribuent à la traçabilité, à la reproductibilité des tests, à la couverture des exigences fonctionnelles, et in fine à l'évaluation de la qualité globale du logiciel.

Dans ce contexte, le développement piloté par les tests (TDD) est une pratique de développement logiciel dans laquelle les cas de test sont élaborés avant l'implémentation du code. Cette méthode

consiste à partir d'une exigence fonctionnelle, à rédiger un cas de test correspondant, puis à développer du code destinée à faire passer ce test. Comme indiqué dans l'article (Teschner, 2020) le processus se répète de manière itérative, renforçant ainsi la compréhension des attentes, la traçabilité des fonctionnalités et la qualité du logiciel produit. La conception des cas de test, dans une logique TDD, ne se limite donc pas à valider un code existant, mais participe activement à la définition du comportement attendu du système.

2.2 Modèles de langage de grande taille (LLM)

Les LLM (modèles de langage de grande taille) sont des modèles d'intelligence artificielle spécifiquement conçus pour comprendre et générer du langage naturel à travers une variété de tâches, telles que la génération de texte, la classification de texte, la traduction automatique, la réponse aux questions, et le résumé automatique (Wei et al., 2022). Ces modèles possèdent un nombre élevé de paramètres dont les poids constituent une partie essentielle, ainsi qu'une architecture d'entraînement leur permettant de réaliser diverses tâches en traitement du langage naturel (NLP). Les recherches ont démontré qu'une augmentation de la taille des modèles améliore leur capacité à accomplir des tâches complexes (Yadong Lu, 2023; Yusheng Su,2023), incitant ainsi les chercheurs à explorer l'impact de cette mise à l'échelle par l'augmentation du nombre de paramètres (Yuan et al., 2023).

Pour différencier les modèles de langage selon leur échelle de paramètres, le terme "modèles de langage de grande taille" (LLM) a été adopté pour désigner les modèles pré-entraînés (PLM) de taille considérable. Les LLM font généralement référence à des modèles possédant des centaines de milliards de paramètres et étant entainés sur des corpus textuels massifs, tels que GPT-3, PaLM, Codex et LLaMA. Ces modèles reposent sur l'architecture Transformer, qui empile plusieurs couches d'attention multi-têtes dans un réseau neuronal profond. Bien que les LLM existants utilisent des architectures similaires (basées sur le Transformer) et des objectifs de pré-entraînement comme la modélisation de langage, ce qui les distingue des modèles plus petits est leur échelle considérablement supérieure en termes de taille du modèle, de données d'entraînement et de puissance de calcul (Fan et al., 2023). Cela leur permet d'avoir une compréhension plus fine du langage naturel et de générer un texte de haute qualité en fonction du contexte ou des invites (Wang, 2024).

Les modèles tels que LLaMA 3B-8B montrent qu'il est possible d'atteindre une haute performance tout en ayant des tailles de paramètres bien inférieures aux centaines de milliards. Ces modèles, bien que plus petits, conservent une capacité impressionnante pour traiter et générer du langage naturel avec efficacité.

2.3 Travaux connexes

Dans cette section, nous examinons les travaux connexes sur la génération manuelle des cas de test, automatisation de cas de test, l'utilisation des LLM en ingénierie logicielle, ainsi que l'application des LLM à la génération de cas de test, afin de mettre en évidence l'importance de notre étude.

2.3.1 Génération des cas de test manuellement

Les tests manuels consistent à ce que les cas de test soient rédigés et exécutés directement par un testeur humain. Cette approche présente plusieurs avantages. D'une part, les testeurs sont capables de percevoir des comportements spécifiques du système qui pourraient échapper à une solution

automatisée. D'autre part, leur flexibilité permet d'adapter les tests en fonction des contextes ou des anomalies rencontrées, ce qui contribue à la viabilité et à la pertinence des tests. Toutefois, le test manuel présente aussi plusieurs limites. Il est souvent long à exécuter, et la rédaction des cas de test demande un effort considérable pour garantir une couverture fonctionnelle suffisante. De plus, cette méthode repose sur l'intervention de nombreuses ressources humaines, ce qui peut engendrer des coûts importants. Elle peut également devenir fastidieuse, monotone et répétitive pour les testeurs. Enfin, certains éléments d'interface graphique (GUI), tels que les variations de taille de pixels ou de combinaisons de couleurs, sont difficiles à repérer manuellement, ce qui peut nuire à la qualité globale du test (Oriat, 2005).

2.3.2 Automatisation de la génération des cas de test

La création de cas de test unitaires vise à valider individuellement le bon fonctionnement des différentes unités ou composants d'un logiciel. Pour chaque méthode cible (souvent désignée comme « méthode principale »), le test associé comprend généralement deux éléments : un préambule de test et un oracle. Le préambule est une suite d'instructions (assignations, appels de méthodes, etc.) destinée à amener la méthode dans un état spécifique à tester. L'oracle, quant à lui, permet de vérifier si le comportement observé correspond aux attentes, généralement à l'aide d'instructions d'assertion.

Afin de réduire le travail manuel nécessaire à la rédaction de ces tests, de nombreuses approches ont été développées pour automatiser leur génération. Diverses approches ont été explorées par les chercheurs pour automatiser cette génération, notamment la recherche guidée (Harman, 2010), (Casamayor, 2023), l'analyse par contraintes (Xiao, 2013), les heuristiques comme Mutation (Jeevarathinam, 2010) ou Oracle heuristique (Hossain, 2023), randomisation (Hall, 2017), les algorithmes basés sur des modèles (Pacheco, 2007), les algorithmes basés sur la recherche en apprentissage profond 91,000.

Néanmoins, ces tests générés automatiquement posent encore des défis en matière de pertinence fonctionnelle et de couverture du logiciel testé.

2.3.3 Génération des cas de test avec LLM

Les modèles de langage de grande taille (LLM) se sont imposés comme des outils performants dans de nombreux domaines liés à la compréhension et à la génération du langage naturel, y compris en ingénierie logicielle (Huang, 2024) ainsi que (Wang, 2024) ont mis en évidence l'utilisation croissante des LLM pour automatiser différentes tâches de test logiciel. Par exemple, TestPilot, un outil de génération de tests basé sur LLM pour JavaScript, capable de produire automatiquement des tests unitaires pour l'ensemble des fonctions API d'un paquet npm.

Malgré les avancées récentes, la majorité des travaux existants en génération automatique de tests se concentrent sur des approches basées soit sur des méthodes Java spécifiques (méthodes focales → cas de test), soit sur des descriptions textuelles seules (texte → cas de test). Ces approches présentent toutefois plusieurs limitations notables. En particulier, elles négligent souvent le contexte fonctionnel et les exigences métier qui entourent la méthode à tester, ce qui conduit à des cas de test parfois incomplets, voire non pertinents. De plus, ces méthodes ne s'alignent pas avec les principes du développement piloté par les tests (TDD), qui impose l'écriture des cas de test à partir des spécifications, avant même le développement du code. Sans appui sur une description textuelle, les modèles ont du mal à capturer les comportements attendus, les cas limites et les scénarios d'erreur,

limitant ainsi la fiabilité des cas générés. Pour répondre à ces défis, nous proposons une approche hybride fondée sur l'utilisation conjointe des descriptions textuelles et des méthodes focales (Texte, Méthodes focales → Cas de test). En exploitant les commentaires présents dans le code, cette méthode permet de produire des cas de test plus riches, précis et représentatifs de l'intention métier, tout en renforçant la cohérence avec les objectifs de validation.

3 LLaMA pour la génération de cas de test

3.1 Création base de données

À ce jour, il n'existe aucune base de données de référence contenant des cas de test « nettoyés », c'est-à-dire des cas de test standardisés et généralisables à différents types d'applications. Pour combler cette lacune, nous avons entrepris la collecte de projets Java open source parmi plus de 91,000 projets disponibles publiquement sur GitHub, dans le but d'extraire les méthodes focales, leurs descriptions textuelles et les cas de test associés.

Cette démarche nous a permis de constituer une base de données de **136060** lignes, chacune composée d'un cas de test, de la description de la méthode cible, et de la méthode elle-même. Comme illustré dans la figure ci-dessus, l'extraction des données a été réalisée à partir de plusieurs dépôts GitHub.

Nous avons développé un script d'analyse parcourant les fichiers source ligne par ligne afin d'identifier :

- les méthodes de test annotées avec @Test (JUnit),
- les commentaires les précédant,interprétés comme descriptions textuelles,
- ainsi que les méthodes Java invoquées dans le corps des tests, considérées comme méthodes focales.

Sur la base de ces éléments, un mappage a été effectué entre chaque cas de test, sa description textuelle, et la méthode ciblée. Ce processus a permis de construire un jeu de données structuré, dans lequel chaque entrée contient une description, une méthode focale et un cas de test associé, facilitant ainsi l'entraînement de modèles de génération automatique de tests.

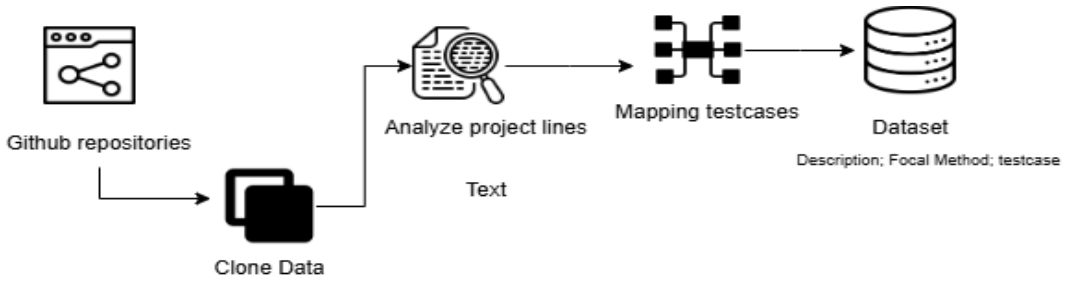

Figure 1: Vue d'ensemble de notre approche de construction de la base de données

Comme illustré dans la figure ci-dessus, le processus d'extraction des données a été réalisé à partir de plusieurs dépôts GitHub. Nous avons développé un script d'analyse qui parcourt les fichiers des projets ligne par ligne, afin d'identifier les méthodes annotées avec @Test (JUnit), leurs descriptions en commentaires, ainsi que les méthodes Java invoquées dans ces tests, considérées comme

méthodes focales. À partir de ces informations, nous avons effectué un mappage entre chaque test, sa description textuelle et la méthode ciblée. Ce processus nous a permis de construire un jeu de données structuré, contenant pour chaque entrée une description, une méthode focale, et un cas de test associé.

La création de cette base de données à grande échelle a nécessité une puissance de calcul importante, notamment en termes de mémoire vive (RAM) et de capacité de traitement. Pour répondre à ces exigences, nous avons opté pour l'utilisation de Google Colab, qui offre un environnement de calcul cloud doté de ressources matérielles performantes (CPU et RAM), facilitant ainsi le traitement massif et parallèle des fichiers issus de milliers de dépôts.

3.2 Génération des cas de test

Une fois la base de données constituée, nous avons sélectionné un ensemble de **7606** jeux de données pour évaluer la génération de cas de test avec **Meta-Llama-3-8B**. Chaque entrée fournit en input la méthode cible ainsi que sa description, tandis que l'output attendu est un cas de test correspondant. Pour interagir avec le modèle Meta-Llama-3-8B, nous avons utilisé l'API de Hugging Face dans le cadre de cette expérimentation.

La qualité de la génération dépend fortement du prompt utilisé : celui-ci doit fournir des indications suffisamment précises pour orienter efficacement le modèle. En effet, sans consignes claires, Meta-Llama-3-8B tend à produire des résultats imprécis ou hors contexte (Sahoo, 2024).

3.2.1 Prompt-engineering

Nous avons exploré plusieurs formulations de prompts, allant de modèles simples à des structures plus complexes et enrichies. Cette démarche nous a permis d'affiner progressivement notre prompt, en intégrant des éléments contextuels et structurants, jusqu'à obtenir une version plus performante.

Nous présentons ici deux types de prompts testés dans le cadre de notre expérimentation :

- Un prompt de base, concis et direct, utilisé en point de départ.
- Un prompt optimisé, fruit de plusieurs itérations successives.

Le prompt initial se limite à une formulation minimale, reposant sur les capacités implicites du modèle à interpréter une consigne générique. Sa forme est la suivante*: "Génère un cas de test JUNIT "*

À partir de cette version de base, nous avons construit un prompt enrichi, affiné à travers une série de tests sur un sous-ensemble de validation restreint (distinct des jeux de données de test et d'entraînement, afin d'éviter toute fuite de données). Nous avons procédé à une analyse qualitative manuelle des résultats générés, et ajusté progressivement le prompt en fonction des erreurs ou insuffisances observées, en dialoguant notamment avec Llama-3-8B pour proposer de nouvelles variantes plus adaptées.

Le prompt final a été pensé pour fournir un cadre détaillé et orienté, en insistant sur plusieurs points clés :

- L'importance de produire des assertions pertinentes et non redondantes,
- Le respect d'une structure claire de cas de test (méthode, annotation @Test, assertions),
- La désignation explicite de la classe et de la méthode ciblée,

Et enfin, l'ajout d'une description contextuelle qui guide la génération.

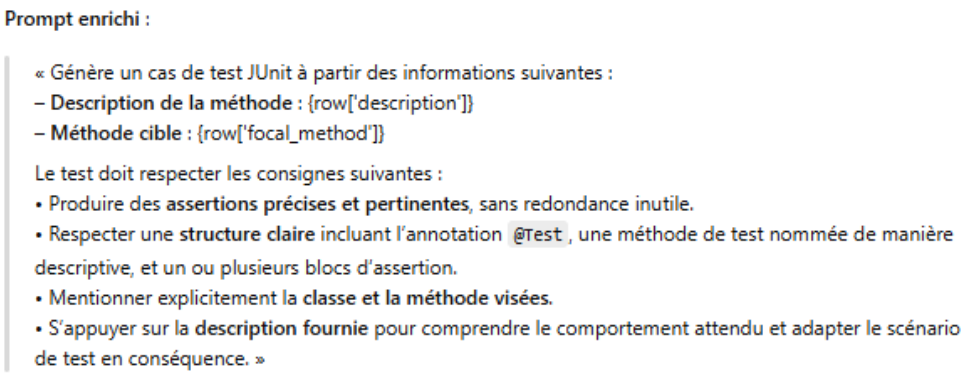

Figure 2: Prompt enrichi pour générer les cas de test

Ce prompt amélioré suit un format structuré, composé de plusieurs blocs :

- **Commentaire introductif :** informe le modèle de l'objectif attendu (générer un cas de test structuré et pertinent).
- **Gabarit de test :** propose un squelette de méthode avec @Test et des emplacements pour les assertions courantes.
- **Nom de classe et de méthode :** indique la cible exacte du test à produire.
- **Description de la méthode :** fournit le contexte fonctionnel pour guider la génération.

3.2.2 LLM-as-a-Judge

Dans le but de compléter cette analyse manuelle et d'évaluer objectivement la qualité des cas de test générés, nous avons également recours à **LLM-as-a-Judge**. Cette méthode innovante exploite la capacité des modèles de langage à évaluer les productions d'autres LLM selon des critères précis définis par l'utilisateur. Elle constitue une alternative efficace à l'évaluation humaine, souvent coûteuse et chronophage. LLM Judge permet notamment :

- L'évaluation d'une sortie unique, avec ou sans référence.
- La comparaison de paires de sorties générées.

En fournissant un critère explicite d'évaluation, le LLM attribue un score en prenant en compte des éléments tels que l'entrée initiale, le contexte de génération, ou encore la pertinence fonctionnelle dans le cas de pipelines RAG. L'essor de cette approche s'explique par les limites des méthodes d'évaluation traditionnelles, qui ne capturent pas toujours la richesse contextuelle et sémantique des résultats générés.

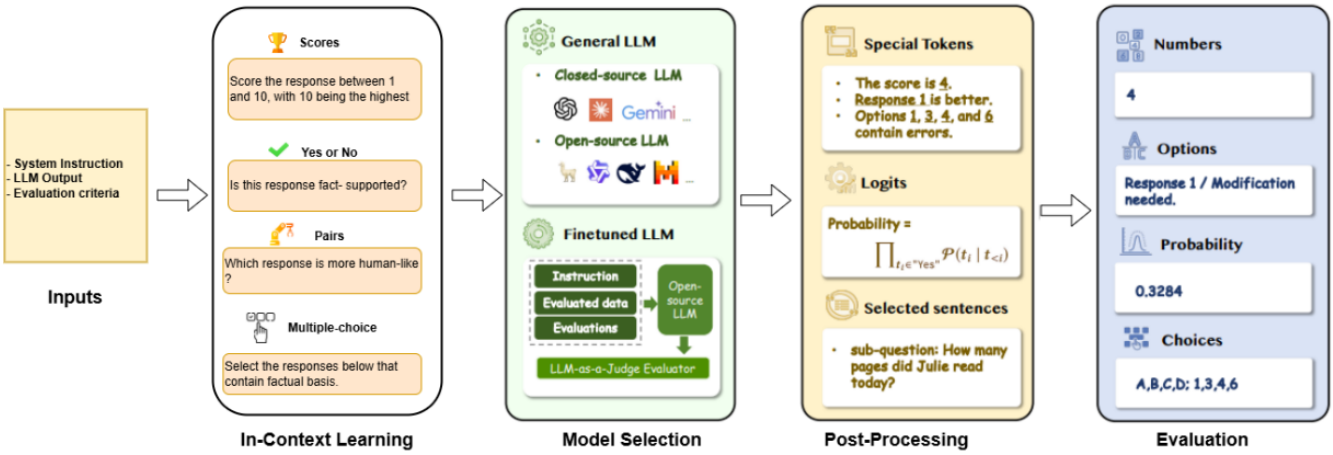

Figure 3: Pipelines d'évaluation par LLM-as-a-Judge

3.2.1.1 In-Context Learning

Pour appliquer la méthode LLM-as-a-Judge, on utilise généralement l'apprentissage par contexte (In-Context Learning), en fournissant au modèle des instructions et des exemples pour guider son raisonnement. Deux éléments sont essentiels :

- La conception des entrées, qui dépend du type de données à évaluer (texte, image, vidéo), de leur mode de présentation (individuelle, par paires, en lot) et de leur position dans le prompt.
- La conception du prompt, qui peut adopter quatre approches : attribution de scores, réponses vrai/faux, comparaisons par paires, ou choix multiples.

3.2.1.2 Model Selection

Pour automatiser l'évaluation dans le cadre de LLM-as-a-Judge, une approche efficace consiste à utiliser des modèles de langage avancés comme GPT-4, en remplacement des évaluateurs humains. Dans notre étude, nous avons opté pour le modèle Llama 3 8B. Des travaux tels que ceux de (Iyengar, 2025) ou (Zheng, 2023) ont montré que GPT-4 peut noter automatiquement des réponses à des questions complexes, avec une précision comparable, voire supérieure, à celle des évaluateurs humains, notamment en termes de cohérence et de stabilité.

3.2.1.3 Post-Processing

Le post-traitement permet d'affiner les distributions de probabilité générées par un LLM utilisé comme juge, afin d'améliorer la précision des évaluations. Il doit être cohérent avec la méthode d'apprentissage par contexte (In-Context Learning) et vise à renforcer la fiabilité des résultats extraits. Trois techniques principales sont utilisées :

- L'extraction de jetons spécifiques,
- La normalisation des logits de sortie,
- La sélection de phrases à forte valeur ajoutée.

Cependant, ces méthodes présentent des limites importantes, notamment pour l'évaluation de questions objectives. Par exemple, une mauvaise extraction du jeton-clé dans une réponse textuelle peut fausser les résultats. Ces difficultés sont étroitement liées à la conception du prompt et à la capacité du modèle à suivre les instructions (Yu, 2024).

3.2.1.4 Evaluation

Une fois les trois étapes principales complétées, on obtient l'évaluation finale notée E. L'ensemble du processus, de l'entrée à la sortie, constitue le pipeline d'évaluation LLM-as-a-Judge, représenté dans la Figure 3.

Une fois les cas de test générés et améliorés grâce aux techniques de prompt engineering et à l'évaluation par LLM-as-a-Judge, une dernière étape consiste à vérifier leur validité syntaxique à l'aide de Tree-sitter.

3.3 Vérification syntaxique des cas de test générés

Afin de vérifier la validité syntaxique des **7606** cas de test générés, nous avons intégré **Tree-sitter** dans un script automatisé dédié à l'analyse. Tree-sitter est un analyseur syntaxique incrémental capable de construire des arbres de syntaxe abstraite (AST) à partir du code source. Cet outil nous a permis d'examiner en temps réel chaque cas de test généré par le modèle **Llama-3-8B**, en identifiant toute erreur de syntaxe ou incohérence structurelle.

Le processus d'analyse repose sur un script automatisé qui parcourt l'ensemble des cas de test produits et valide leur conformité avec les conventions **JUnit** ainsi qu'avec la grammaire du langage Java. Parmi les critères de validation figurent notamment la présence d'annotations appropriées (telles que @Test), la structure correcte des classes, et le respect des standards de nommage. En cas d'anomalie, Tree-sitter génère des rapports détaillés, facilitant l'identification et la correction des erreurs.

L'évaluation de l'ensemble des cas de test a révélé qu'environ **3%** d'entre eux présentaient des erreurs syntaxiques. Les types d'erreurs les plus fréquemment observés sont les suivants : Espaces indésirables dans les noms de méthodes, en contradiction avec les règles de nommage de Java, rendant le code non compilable. Tests vides, c'est-à-dire des méthodes de test ne contenant ni actions ni assertions. Par ailleurs, d'autres anomalies structurelles ont été relevées, telles que des problèmes de fermeture de guillemets, des accolades mal fermées, l'insertion de phrases en français directement dans le corps du code sans commentaire, ainsi que des erreurs de formatage liées à l'utilisation de balises Markdown (java) intégrées par inadvertance dans les blocs de code Java.

```
import org.junit.Test;
import org.junit.Assert;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
import java.sql.Connection;
import java.sql.Statement;

public class TestCase73 {
      @Test
   public void testAddDom SuccessfulAddDom() {
      // Given
      WXSDKInstance instance = mock(WXSDKInstance.class);
      WXRegistry registry = mock(WXRegistry.class);
      WXRenderManager renderManager = mock(WXRenderManager.class);
      WXEnvironment environment = mock(WXEnvironment.class);
      WXDomStatement statement = new WXDomStatement();
      JSONObject dom = new JSONObject();
      String parentRef = "";
      int index = 0;

      // When
      statement.addDom(dom, parentRef, index);

      // Then
      verify(instance, times(1)).commitUTStab(IWXUserTrackAdapter.DOM_MODULE, WXErrorCode.WX_SUCCESS);
      verify(renderManager, times(1)).addComponent(meq(instance.getInstanceId()), anyString(), anyInt());
      verify(environment, times(1)).isApkDebugable();
   }
}
```

Figure 4: exemple d'erreur syntaxique — présence d'un espace dans le nom de la méthode

```
import org.junit.Test;
import org.junit.Assert;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
import java.sql.Connection;
import java.sql.Statement;

public class TestCase91 {
      @Test
   public void testShouldResetDivId() {
      // Initialisation de la méthode
      int idCacheSlots = 2;
      AtomicLong idCacheSlotsAtomicLong = new AtomicLong(idCacheSlots);
      ConcurrentHashMap<String, String> seenCharts = new ConcurrentHashMap<>();

      // Cas 1 : divId nouveau, idCacheSlots > 0
      boolean result1 = ShouldResetDivId.shouldResetDivId("divId1");
      assertEquals(false, result1);
      assertEquals(1, idCacheSlotsAtomicLong.get());
      assertEquals("divId1", seenCharts.get("divId1"));

      // Cas 2 : divId déjà vu, idCacheSlots > 0
      boolean result2 = ShouldResetDivId.shouldResetDivId("divId1");
      assertEquals(true, result2);
      assertEquals(1, idCacheSlotsAtomicLong.get());
      assertEquals("divId1", seenCharts.get("divId1"));

      // Cas 3 : divId nouveau, idCacheSlots <= 0
      idCacheSlotsAtomicLong.set(0);
      boolean result3 = ShouldResetDivId.shouldResetDivId("divId2");
      assertEquals(true, result3);
      assertEquals(0, idCacheSlotsAtomicLong.get());
      assertEquals("divId2", seenCharts.get("divId2"));

      // Cas 4 : divId déjà vu, idCacheSlots <= 0
      boolean result4 = ShouldResetDivId.shouldResetDivId("divId2");
      assertEquals(true, result4);
      assertEquals(0, idCacheSlotsAtomicLong.get());
      assertEquals("divId2", seenCharts.get("divId2"));
   }
}
```

Figure 5: Exemple d'erreur syntaxique : guillemet manquant autour de divId2

```
import org.junit.Test;
import org.junit.Assert;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
import java.sql.Connection;
import java.sql.Statement;

public class TestCase1 {
   @Test
public void testPrepareCallWithSqlText() {
   Object proxy = new Object();
   Method method = Connection.class.getMethod("prepareCall", String.class);
   Object[] args = new Object[] { "SELECT * FROM users WHERE id =?" };

   try {
      Object result = invoke(proxy, method, args);
      Assert.assertTrue(result instanceof Statement);
      Statement stmt = (Statement) result;
      Assert.assertTrue(stmt instanceof Proxy);
      // Verify the SQL text is annotated with the trace span
      // TO DO: implement the verification logic here
   } catch (Throwable e) {
      Assert.fail("Unexpected exception: " + e.getMessage());
   }
}
}
```

Figure 6: Cas de test sans erreurs syntaxique

4 Conclusion et perspectives

Dans cet article, nous proposons une approche de génération de cas de test à partir de texte et de méthode focale, s'appuyant sur un modèle de langage de grande taille. Cette méthode combine des techniques de prompt engineering et une évaluation automatique via la stratégie LLM-as-a-Judge. À l'issue de ce travail, nous avons constitué une base de données comprenant 136 060 lignes, chacune composée d'une méthode focale, de sa description et du cas de test généré correspondant.

En perspective, nous prévoyons d'effectuer un fine-tuning du modèle ainsi qu'une nouvelle évaluation des cas de test générés, en y intégrant notamment la mesure de couverture de code. Nous envisageons également de renforcer cette couverture en recourant à des techniques de test par mutation.

Notre étude d'ablation révèle une amélioration significative des performances grâce à l'usage combiné d'un prompt design optimisé et d'une évaluation LLM-as-a-Judge adaptée au modèle LLaMA 3.8B, soulignant l'importance de ces composants pour la génération automatique de cas de test à partir de texte. Sur la base de nos résultats, nous formulons deux recommandations principales : (1) l'utilisation basique du modèle LLaMA 3.8B, sans prompt structuré, ne permet pas d'atteindre des performances satisfaisantes dans cette tâche ; et (2) un prompt efficace doit systématiquement être intégré pour exploiter pleinement les capacités d'un LLM.

Enfin, malgré les améliorations apportées par LLaMA 3.8B, des erreurs syntaxiques persistent dans environ 3 % des cas générés. Parmi les plus fréquemment observées, on relève : des espaces indésirables dans les noms de méthodes (non conformes aux conventions Java), l'absence de guillemets dans les chaînes de caractères, des assertions vides sans conditions de vérification, ou encore des erreurs de fermeture de blocs de code, rendant ainsi les tests non compilables.

5 Références

Wei, J., Tay, Y., Bommasani, R., Raffel, C., Zoph, B., Borgeaud, S., Yogatama, D., Bosma, M., Zhou, D., Metzler, D., et al., 2022.Emergent abilities of large language models.arXiv preprint arXiv:2206.07682 .

Casamayor, R. a. C. C. a. P. s. a. P. r. F. (2023). Studying the Influence and Distribution of the Human Effort in a Hybrid Fitness Function for Search-Based Model-Driven Engineering. *IEEE Trans. Softw. Eng.*, *49*(12), 5189–5202 , numpages = 5114. https://doi.org/10.1109/TSE.2023.3329730

Cummins, C. a. S. V. a. G. D. a. R. B. a. G. J. a. S. G. a. L. H. (2024). Meta Large Language Model Compiler: Foundation Models of Compiler Optimization. https://doi.org/10.48550/arXiv.2407.02524

Deng, Y. a. X. C. a. P. H. a. Y. C. a. Z. L. (2023, 07). *Large Language Models Are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models*

Yusheng Su, Chi-Min Chan, Jiali Cheng, Yujia Qin, Yankai Lin, Shengding Hu, Zonghan Yang, Ning Ding, Xingzhi Sun, Guotong Xie, Zhiyuan Liu, Maosong Sun (2023,06). *Exploring the Impact of Model Scaling on Parameter-Efficient Tuning.*

formés (2023,09). *An Empirical Study of Scaling Instruction-Tuned Large Multimodal Models.*

Dong, Y. a. J. X. a. J. Z. a. L. G. (2023). Self-collaboration Code Generation via ChatGPT. https://doi.org/10.48550/arXiv.2304.07590

Yuan, Z., Lou, Y., Liu, M., Ding, S., Wang, K., Chen, Y., Peng, X., 2023.No more manual tests? evaluating and improving chatgpt for unit test generation.arXiv preprint arXiv:2305.04207 .

Endres, M. a. F. S. a. C. S. a. L. S. K. (2024). Can Large Language Models Transform Natural Language Intent into Formal Method Postconditions? *Proc. ACM Softw. Eng.*, *1*(FSE). https://doi.org/10.1145/3660791

Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, Jie M. Zhang. (2023, 05). *Large Language Models for Software Engineering: Survey and Open Problems*

Gong, J. a. V. V. a. B. P. a. W. F. a. J. W. a. X. J. a. G. R. a. B. M. a. K. L. a. W. Z. (2025). Language Models for Code Optimization: Survey, Challenges and Future Directions. https://doi.org/10.48550/arXiv.2501.01277

Hall. (2017). *SealTest: a simple library for test sequence generation* Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, https://doi.org/10.1145/3092703.3098229

Harman, M. a. M. P. (2010). A Theoretical and Empirical Study of Search-Based Testing: Local, Global, and Hybrid Search. *Software Engineering, IEEE Transactions on*, *36*, 226 - 247. https://doi.org/10.1109/TSE.2009.71

Hossain, S. B. a. F. A. a. D. M. B. a. E. S. a. V. W. (2023). *Neural-Based Test Oracle Generation: A Large-Scale Evaluation and Lessons Learned* Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, https://doi.org/10.1145/3611643.3616265

Hou, X. a. Z. Y. a. L. Y. a. Y. Z. a. W. K. a. L. L. a. L. X. a. L. D. a. G. J. a. W. H. (2023). Large Language Models for Software Engineering: A Systematic Literature Review. https://doi.org/10.48550/arXiv.2308.10620

Huang, K. a. M. X. a. Z. J. a. L. Y. a. W. W. a. L. S. a. Z. Y. (2024). *An Empirical Study on Fine-Tuning Large Language Models of Code for Automated Program Repair* Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering, https://doi.org/10.1109/ASE56229.2023.00181

Iyengar, A. a. K. A. a. K. R. a. M. S. (2025). A Generative Caching System for Large Language Models. https://doi.org/10.48550/arXiv.2503.17603

Jeevarathinam, M. a. T. A. (2010). Test Case Generation using Mutation Operators and Fault Classification. *International Journal of Computer Science and Information Security*, *7*.

Li, J. a. Z. Y. a. L. Y. a. L. G. a. J. Z. (2023). Towards Enhancing In-Context Learning for Code Generation. https://doi.org/10.48550/arXiv.2303.17780

Liu, J. a. C. Y. a. L. M. a. P. X. a. L. Y. (2024). STALL+: Boosting LLM-based Repository-level Code Completion with Static Analysis. https://doi.org/10.48550/arXiv.2406.10018

Liu, J. a. X. C. a. W. Y. a. Z. L. (2023). Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. https://doi.org/10.48550/arXiv.2305.01210

Oriat, C. (2005). Jartege: A Tool for Random Generation of Unit Tests for Java Classes. *Lecture Notes in Computer Science*. https://doi.org/10.1007/11558569_18

Pacheco, C. a. E. M. (2007, 10). *Randoop: Feedback-directed random testing for Java* Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA,

Poldrack, R. a. L. T. a. B. G. (2023). AI-assisted coding: Experiments with GPT-4. https://doi.org/10.48550/arXiv.2304.13187

Sahoo, P. a. S. A. a. S. S. a. J. V. a. M. S. a. C. A. (2024). A Systematic Survey of Prompt Engineering in Large Language Models: Techniques and Applications. https://doi.org/10.13140/RG.2.2.13032.65286

Teschner, T.-R. (2020). A practical guide towards agile test-driven development for scientific software projects.

Wang, J. a. H. Y. a. C. C. a. L. Z. a. W. S. a. W. Q. (2024). Software Testing With Large Language Models: Survey, Landscape, and Vision. *IEEE Transactions on Software Engineering*, *PP*, 1-27. https://doi.org/10.1109/TSE.2024.3368208

Xiao, X. a. L. S. a. X. T. a. T. N. (2013). *Characteristic studies of loop problems for structural test generation via symbolic execution* Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering, https://doi.org/10.1109/ASE.2013.6693084

Yetiştiren, B. a. Ö. I. a. A. M. a. T. E. (2023). Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT. https://doi.org/10.48550/arXiv.2304.10778

Yu, Q. a. Z. Z. a. S. S. a. L. Z. a. X. F. a. T. B. a. C. D. (2024). xFinder: Robust and Pinpoint Answer Extraction for Large Language Models. https://doi.org/10.48550/arXiv.2405.11874

Zheng, L. a. C. W.-L. a. S. Y. a. Z. S. a. W. Z. a. Z. Y. a. L. Z. a. Z. L. a. L. D. a. X. E. a. (2023). Judging LLM-as-a-judge with MT-Bench and Chatbot Arena. https://doi.org/10.48550/arXiv.2306.05685