Amélioration des techniques de génération de traces d'exécution sur des applications Android pour la détection dynamique de défauts de code

par

Houcine Abdelkader CHERIEF

MÉMOIRE PRÉSENTÉ À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
COMME EXIGENCE PARTIELLE À L'OBTENTION DE LA MAÎTRISE
AVEC MÉMOIRE EN GÉNIE LOGICIEL
M. Sc. A.

MONTRÉAL, LE 23 MARS 2026

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

PRÉSENTATION DU JURY

CE MÉMOIRE A ÉTÉ ÉVALUÉ

PAR UN JURY COMPOSÉ DE:

Mme. Naouel MOHA, directrice de mémoire
Département de génie logiciel et TI, École de technologie supérieure

M. Florent AVELLANEDA, codirecteur
Département d'informatique, Université de Québec à Montréal

M. Christopher FUHRMAN, président du jury
Département de génie logiciel et TI, École de technologie supérieure

M. Sègla Jean-Luc KPODJEDO, membre du jury
Département de génie logiciel et TI, École de technologie supérieure

IL A FAIT L'OBJET D'UNE SOUTENANCE DEVANT JURY ET PUBLIC

LE 17 MARS 2026

À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

REMERCIEMENTS

Tout d'abord, je souhaite remercier Dieu pour Sa grâce et Son soutien tout au long de ce parcours. Il a été une source de force et d'inspiration dans les moments difficiles et de doute. Je Le remercie de m'avoir donné l'opportunité et la capacité de mener à bien ce projet.

Je dédie ce travail à mes parents, qui ont toujours cru en moi et n'ont jamais ménagé leurs efforts pour m'aider et me soutenir dans mes choix.

Je souhaite exprimer ma profonde gratitude à mes directeurs de mémoire, Mme Naouel MOHA et M. Florent AVELLANEDA, pour leur encadrement, leurs conseils et leur soutien tout au long de ce travail. Je tiens à les remercier pour leurs encouragements, leur confiance et leur expertise, ainsi que pour leurs idées et solutions qui ont permis de surmonter les différentes difficultés rencontrées. Leurs commentaires pertinents et leurs critiques constructives ont grandement contribué à améliorer la qualité de ce travail.

Je tiens également à exprimer mes sincères remerciements à toutes les personnes ayant contribué, notamment M. Quentin STIÉVENART, ainsi que mes amis Brahim MAHMOUDI et Zacharie CHENAIL-LARCHER, pour leur aide précieuse et leur collaboration.

Enfin, je remercie les membres du jury pour la lecture attentive de ce mémoire. J'apprécie l'intérêt qu'ils ont porté ainsi que leurs commentaires et suggestions constructifs, qui ont contribué à en améliorer la qualité.

Amélioration des techniques de génération de traces d'exécution sur des applications Android pour la détection dynamique de défauts de code

Houcine Abdelkader CHERIEF

RÉSUMÉ

Les applications mobiles sont devenues essentielles dans notre vie quotidienne, ce qui fait de la qualité du code une préoccupation critique pour les développeurs. Les défauts de code comportementaux sont des caractéristiques présentes dans le code source qui induisent un comportement inapproprié lors de l'exécution, affectant négativement la qualité logicielle en termes de performance, de consommation d'énergie et d'utilisation de la mémoire.

L'analyse dynamique s'est révélée efficace pour la détection des défauts de code comportementaux dans les applications Android. Bien qu'elle surpasse les outils d'analyse statique, elle nécessite une couverture élevée des événements liés à ces défauts, définis comme des instructions ou des appels de méthodes spécifiques associés à des comportements inappropriés du code. En pratique, cette limitation se traduit par un taux élevé de faux négatifs, laissant de nombreuses instances de défauts de code non détectées.

Les grands modèles de langage (LLM) ont récemment démontré des avancées significatives dans divers domaines de recherche et offrent un potentiel prometteur pour la génération de traces d'exécution *intelligentes*, en particulier pour la détection des défauts de code comportementaux dans les applications Android. La *trace d'exécution intelligente* désigne une séquence d'événements générée par des actions ciblées visant à déclencher et observer des comportements spécifiques de l'application.

Ce mémoire propose les principales contributions suivantes : (1) DynamicsLLM, un outil de l'analyse dynamique, qui exploite les LLM afin de générer intelligemment des traces d'exécution. (2) Une approche *hybride* conçue pour améliorer la couverture des événements liés aux défauts de code, en particulier pour les applications comportant un nombre limité d'activités Android. (3) Une validation de DynamicsLLM sur 307 applications mobiles libres accès provenant de F-DROID, incluant une comparaison avec le dernier outil de l'état de l'art Dynamics. Les résultats montrent que, sous un nombre limité d'actions sur l'interface, DynamicsLLM utilisant uniquement les LLM couvre jusqu'à trois fois plus d'événements liés aux défauts de code que Dynamics. L'approche *hybride* améliore la couverture des LLM de 22.6% pour les applications contenant peu d'activités. De plus, 14% des événements liés aux défauts de code qui ne peuvent pas être déclenchés par Dynamics sont effectivement déclenchés par DynamicsLLM. (4) Enfin, une stratégie d'exploration fondée sur l'UCB (Upper Confidence Bound) est introduite, qui obtient 12,9% de couverture supplémentaire par rapport à l'hybride sur les applications de taille moyenne, tout en offrant des performances compétitives pour les autres catégories.

Mots-clés: Analyse dynamique, défauts de code comportementaux, grands modèles de langage, génération de traces d'exécution.

Improving execution-trace generation techniques for Android applications for the dynamic detection of code smells

Houcine Abdelkader CHERIEF

ABSTRACT

Mobile apps are essential to our daily lives, making code quality a critical concern for developers. Behavioural code smells are characteristics in the source code that induce inappropriate code behaviour during execution, which negatively impact software quality in terms of performance, energy consumption, and memory.

Dynamic analysis has proven effective for detecting behavioral code defects in Android applications. Although it outperforms static analysis tools, it requires high coverage of code smell-related events, defined as specific instructions or method calls associated with inappropriate code behavior. In practice, this limitation results in a high rate of false negatives, leaving many instances of code defects undetected.

Large Language Models (LLM) have achieved notable advances across numerous research domains and offer significant potential for generating *intelligent* execution traces, particularly for detecting behavioural code smells in Android mobile applications. By *intelligent execution trace*, we mean a sequence of events generated by specific actions in a way that triggers the identification of a given behaviour.

In this work, the main contributions are : (1) DynamicsLLM, an enhanced tool built upon the state-of-the-art Dynamics approach, which leverages LLM to intelligently generate execution traces. (2) A *hybrid* approach designed to improve the coverage of code smell–related events, particularly for applications with a few number of activities. (3) A comprehensive evaluation of DynamicsLLM on 307 mobile applications from the open-source F-DROID dataset, including a comparison with the Dynamics tool. Our results show that, under a limited number of GUI actions, DynamicsLLM configured with 100% LLM covers up to three times more code smell–related events than Dynamics. The *hybrid* approach further improves LLM coverage by 22.6% for applications with a small number of activities. Moreover, 14% of code smell–related events that cannot be triggered by Dynamics are successfully triggered by our tool. (4) Finally, we introduce an exploration strategy based on UCB (Upper Confidence Bound), which achieves 12.9% additional coverage compared to the *hybrid* on medium-sized apps, while keeping competitive performance for the other categories.

Keywords: Dynamic Analysis, Behavioural Code Smells, Large Language Models, Execution Traces Generation.

TABLE DES MATIÈRES

Page

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES ABRÉVIATIONS, SIGLES ET ACRONYMES

LLM	Grands Modèles de Langages
UCB	Upper Confidence Bound
AUT	Application sous le test

INTRODUCTION

Le contexte nécessaire à la compréhension du sujet, les motivations, la problématique ayant conduit à l'élaboration de ce mémoire, ainsi que les principales contributions, sont présentés dans ce chapitre.

Contexte

Les applications mobiles sont devenues une partie essentielle de notre quotidien, avec des milliards de téléchargements depuis des plateformes telles que Apple App Store (statista.com, 2022a) et Google Play Store (statista.com, 2022b) chaque année. Les défauts de code sont définis comme « des mauvaises pratiques de conception et de mise en œuvre de logiciels résultant de mauvais choix d'implémentation ou de conception» (Fowler, 1999). Ils ne sont pas nécessairement des bogues, mais ils peuvent ralentir le développement, augmenter le risque d'erreurs et rendre le code plus difficile à maintenir.

Par conséquent, la détection des défauts de code comportementaux au sein des systèmes logiciels est une priorité importante pour réduire la dette technique (Prestat, Moha, Villemaire & Avellaneda, 2024). En plus des défauts de code orientés objet courants, les applications mobiles ont leurs propres défauts en raison des limitations et contraintes des ressources, telles que la mémoire, les performances et la consommation d'énergie (Prestat, Moha & Villemaire, 2022). Certains de ces défauts sont des défauts de code comportementaux, qui sont des caractéristiques dans le code source qui provoquent un comportement inapproprié pendant l'exécution. Le terme « comportement » fait spécifiquement référence au comportement d'exécution, c'est-à-dire une occurrence ou une séquence d'événements ou d'actions de code observables pendant l'exécution (Prestat *et al.*, 2022).

Pour aborder ce problème, de nombreux outils ont été développés pour détecter les défauts de code comportementaux dans les applications mobiles, en s'appuyant sur des techniques

d'analyse statique, telles que ADoctor (Palomba, Di Nucci, Panichella, Zaidman & De Lucia, 2017) et Paprika (Hecht, Rouvoy, Moha & Duchien, 2015). Prestat *et al.* (2022) ont mené une analyse empirique qui a conclu que ces outils présentent une faible précision, en raison de leur dépendance exclusive à l'analyse statique du code.

Prestat *et al.* (2024) ont proposé une approche dynamique outillée appelée Dynamics. Elle instrumente les applications testées pour collecter des traces d'exécution en temps réel, qui sont ensuite utilisées pour détecter les défauts de code comportementaux. Cette méthode est plus efficace que l'analyse statique en raison de la complexité et de la nature dynamique des applications mobiles, atteignant une précision moyenne de 92,8%. Cependant, le rappel moyen n'est que de 53,4%. L'efficacité de la méthode Dynamics dépend de la qualité de la séquence générée d'événements. La plupart des méthodes d'analyse dynamique utilisent des stratégies de génération aléatoire d'événements, entraînant un grand nombre de faux négatifs (Guo, Qi, Li & Wu, 2024) et une faible couverture de test (Liu *et al.*, 2023), et sont incapables de localiser rapidement le code natif (Cao, Guo & Qu, 2024).

Problématique

Les outils de test automatisés d'interface graphique, tels que MonkeyRunner (Developers, 2010) et Droidbot (Li, Yang, Guo & Chen, 2017), sont largement utilisés pour éviter les tests manuels chronophages et fastidieux. Ces outils explorent les applications et effectuent diverses actions, telles que le défilement et le clic, pour obtenir une trace d'exécution. Cependant, la génération de traces d'exécution à l'aide de ces outils entraîne une couverture limitée du code. Par exemple, il est difficile pour MonkeyRunner d'atteindre certaines activités et de déclencher des événements spécifiques uniquement par des événements aléatoires. De plus, il est encore plus difficile, voire impossible, de passer certaines activités comme la connexion et l'inscription, qui nécessitent de remplir des formulaires avec des formats de données exacts. Cela entraîne

un manque de couverture de code, laissant des événements pertinents, tout en manquant de nombreux faux négatifs, qui sont des défauts de code non détectés mais auraient dû l'être.

Cependant, une étude récemment publiée (Akinotcho, Wei & Rubin, 2024) indique que même les techniques les plus sophistiquées n'atteignent qu'environ 30% de couverture pour les applications du monde réel. Ces outils rencontrent plusieurs défis, notamment des difficultés à générer des *entrées sémantiquement significatives* (comme le formatage correct des mots de passe dans les formulaires d'inscription), et à effectuer des actions sur l'interface graphique dans la *séquence correcte* (par exemple, exécuter des actions dans l'ordre requis pour contourner l'activité actuelle).

En conséquence, les techniques actuelles restent limitées en termes de couverture des événements liés aux défauts de code, c'est-à-dire des instructions ou des appels de méthodes spécifiques associés aux défauts de code comportementaux. Ainsi, une grande partie de ces événements n'est pas déclenchée durant l'exécution, ce qui entraîne une augmentation du nombre de défauts non détectés et, par conséquent, une diminution du rappel de l'analyse dynamique.

Motivation

Les grands modèles de langage pré-entraînés (LLM) ont récemment émergé comme une technologie révolutionnaire dans le traitement du langage naturel et l'intelligence artificielle, démontrant des performances exceptionnelles dans diverses tâches (Wang *et al.*, 2024). De nombreux efforts de recherche (Liu *et al.*, 2023; Wen, Wang, Liu & Li, 2024; Yoon, Feldt & Yoo, 2024) se concentrent sur l'amélioration des tâches de test de logiciels, en particulier dans la génération de traces d'exécution, parmi lesquelles les LLM sont les plus prometteurs. L'approche de test des applications mobiles en utilisant les LLM est présentée comme une tâche de questions-réponses, où l'agent sollicite le LLM pour la prochaine action, et le LLM répond avec les informations nécessaires pour compléter la tâche. Cette approche vise à créer des *traces*

d'exécution intelligentes, qui sont des séquences d'événements générés par des actions spécifiques pour déclencher l'identification d'un comportement donné. Ces traces pourraient aider les outils d'analyse dynamique à détecter plus efficacement les défauts de code comportementaux, en réduisant les faux négatifs. Cependant, l'utilisation de ces générateurs de traces basés sur LLM n'a pas encore été appliquée.

Objectif de la recherche

Par conséquent, ce mémoire vise à combler cette lacune de recherche en améliorant la détection des défauts de code comportementaux Android par l'analyse dynamique, grâce à la production de traces d'exécution intelligentes à l'aide de générateurs basés sur des LLM et par proposer d'autres techniques améliorant la couverture des événements liés aux défauts de code durant l'exécution. L'objectif est ainsi de réduire le nombre de faux négatifs, c'est-à-dire les défauts qui devraient être détectés mais ne le sont pas.

Contributions

Ce travail a donné lieu à une publication dans la conférence FORGE (AI Foundation Models and Software Engineering) colocalisée avec ICSE 2026 (Cherief *et al.*, 2026a).

Les principales contributions sont les suivantes :

- Tout d'abord, un outil entièrement automatique est proposé, appelé DynamicsLLM, qui exploite des LLM librement accessibles afin de générer des traces d'exécution intelligentes, permettant la détection des défauts de code comportementaux.
- Deuxièmement, une nouvelle approche *hybride* est proposée visant à améliorer les performances des LLM en couvrant un plus grand nombre d'événements liés aux défauts de code dans un délai d'une heure, en particulier pour les applications mobiles comportant un nombre limité d'activités.

- Troisièmement, l'outil DynamicsLLM est évalué à l'aide de plusieurs métriques (précision, rappel et couverture des événements liés aux défauts de code) sur un sous-ensemble de 307 applications mobiles, issu du même jeu de données de validation que celui utilisé par Dynamics libres accès provenant de F-DROID. Ensuite, les performances de DynamicsLLM sont comparées à celles de Dynamics, la méthode outillée la plus récente. Les résultats montrent que, avec un nombre limité d'actions sur l'interface graphique, DynamicsLLM, utilisant une approche 100% LLM, couvre jusqu'à trois fois plus d'événements liés aux défauts de code que Dynamics. De plus, l'approche *hybride* améliore la couverture des LLM de 22.6% pour les applications comportant quatre activités ou moins, représentant 63% du jeu de données. Enfin, 14% des événements liés aux défauts de code qui ne peuvent pas être déclenchés par Dynamics sont effectivement déclenchés par DynamicsLLM.
- Bien que l'approche *hybride* présente d'excellentes performances empiriques, elle repose sur des heuristiques définies manuellement. Une autre approche hybride fondée sur l'algorithme d'apprentissage par renforcement UCB (Upper Confidence Bound) constitue une alternative théoriquement justifiée, capable d'adapter automatiquement la stratégie d'exploration à la complexité de l'application. Cette approche permet non seulement d'expliquer, mais aussi d'améliorer la couverture observée, en particulier pour les applications de taille moyenne (12,9% de plus), tout en maintenant des performances compétitives pour les autres catégories.

Paquets de réplication

Tous les outils, les données et les résultats des expérimentations sont disponibles sur :

- Le paquet de réplication de l'article (Cherief, Avellaneda & Moha, 2025) pour des recherches et analyses supplémentaires. Il contient plus de 1 230 heures de traces d'exécution.
- Le paquet de réplication pour les résultats de l'approche UCB contenant 436 heures supplémentaires (Cherief, Avellaneda & Moha, 2026b).

Organisation du mémoire

Ce mémoire se divise en quatre chapitres. Le chapitre 1 passe en revue les travaux connexes regroupés selon différents axes de recherche. Le chapitre 2 présente les informations de base sur les défauts de code comportementaux ainsi que l'outil DroidAgent utilisé pour l'implémentation de l'outil DynamicsLLM. Le chapitre 3 détaille l'approche proposée et son implémentation pour la détection des défauts de code comportementaux et les différentes approches de génération de trace d'exécution étudiées. Le chapitre 4 présente la méthodologie d'évaluation, les résultats obtenus et les limitations identifiées. Ce mémoire est terminé par une conclusion et les travaux futurs.

CHAPITRE 1

ÉTAT DE L'ART

Ce chapitre présente les travaux de la littérature en lien avec ce mémoire. Plus précisément, l'état de l'art concernant : 1) les défauts de code propres aux applications mobiles, 2) les approches de détection existantes, 3) l'analyse dynamique appliquée aux applications mobiles, 4) les méthodes de génération de traces d'exécution, 5) l'analyse dynamique pour la détection des défauts de code et 6) l'utilisation des LLM pour leur identification.

1.1 Les défauts de code spécifiques aux applications mobiles

La détection des défauts de code dans les applications mobiles a été largement discutée dans la littérature. Reimann, Brylski & Aßmann (2014) présentent un catalogue de 30 défauts de qualité pour Android, dérivés de pratiques documentées et d'expériences de développeurs. Ces défauts affectent divers aspects tels que l'implémentation, les interfaces utilisateur et l'utilisation des bases de données, impactant négativement l'efficacité, l'expérience utilisateur et la sécurité. Ghafari, Gadient & Nierstrasz (2017) ont identifié 28 défauts de sécurité indiquant des vulnérabilités potentielles et ont développé un outil d'analyse statique pour la détection. La détection de ces défauts repose sur la présence de certains attributs ou appels de méthode, plutôt que sur un comportement inapproprié du code. La communauté Android continue d'identifier de nouveaux types de défauts pour améliorer la qualité des applications (Wu, Chen & Lee, 2023), comme les 7 défauts comportementaux spécifiques à Android identifiés par (Prestat *et al.*, 2022).

1.2 Approches de détection des défauts de code

De nombreuses approches capturent diverses caractéristiques pour détecter les défauts de code en utilisant différentes techniques d'analyse dans les applications Android (Wu *et al.*, 2023). La plupart des études utilisées pour détecter les défauts comportementaux spécifiques à Android sont des extensions d'analyse statique de ADoctor (Palomba *et al.*, 2017) et Paprika (Hecht *et al.*, 2015). Les deux outils utilisent des techniques de recherche de termes et de calcul

de métriques. Les défauts de code spécifiques à Kotlin (Novendra & Danar Sunindyo, 2024) ont également été explorés, mais les outils existants sont limités à la détection de seulement trois types de défauts de code. Les approches d'apprentissage automatique, telles que celles utilisées pour Deep Wakelock (Khan, Lee, Abbas, Abbas & Bashir, 2021a), un type spécifique de défaut comportemental, ont montré un potentiel, mais nécessitent des ensembles de données volumineux et équilibrés. De plus, des modèles de Perceptron Multicouches (MLP) (Khan, Lee, Wu & Abbas, 2021b) ont été appliqués pour détecter les défauts de Wakelock, bien qu'ils n'abordent pas d'autres fuites de ressources dans les applications mobiles.

Les approches hybrides (Alzaylaee, Yerima & Sezer, 2017; Manukulasooriya *et al.*, 2024; Tsutano, Bachala, Srisa-an, Rothermel & Dinh, 2019) sont utilisées pour détecter les défauts de code en abordant des aspects spécifiques que d'autres méthodes peuvent négliger. Ces approches intègrent à la fois l'analyse statique et dynamique pour améliorer le processus de détection. Cependant, bien que des méthodes comme le test basé sur l'aléatoire puissent rapidement générer des entrées et soient efficaces, elles échouent souvent à explorer toutes les voies d'exécution possibles, risquant ainsi de manquer du code malveillant qui ne s'exécute que dans certaines conditions. Pour remédier à cela, (Yunmar, Kusumawardani, Widyawan & Mohsen, 2024) suggèrent d'utiliser la génération d'entrées contextuelles, qui adapte les entrées au contexte de l'application, augmentant ainsi les chances de déclencher et d'identifier un comportement malveillant. Cela s'aligne spécifiquement avec l'intention de l'approche de ce travail.

1.3 L'analyse dynamique pour les applications mobiles

L'analyse dynamique est utilisée à diverses fins pour traiter les problèmes des applications mobiles. Par exemple, PredRacer (Guo *et al.*, 2024) utilise l'analyse dynamique pour détecter les courses de données potentielles. Cette approche aide à élargir le champ de recherche et à réduire les faux négatifs, tout en minimisant les faux positifs en intégrant les relations de précédence spécifiques au modèle de concurrence Android. L'analyse dynamique est également largement utilisée en sécurité ; par exemple, JNFuzz-Droid (Cao *et al.*, 2024) applique l'analyse dynamique pour le *fuzz testing* et l'analyse des traces afin d'identifier les vulnérabilités dans le

code natif Android en observant le comportement de l'application pendant son exécution. De plus, DaPANDA (Liu *et al.*, 2019) introduit une approche *hybride* pour détecter automatiquement les notifications agressives dans les applications Android. En utilisant une méthode de test guidée et en instrumentant la plateforme Android, DaPANDA identifie efficacement les notifications agressives, améliorant ainsi de manière significative l'expérience utilisateur sur la plateforme mobile.

1.4 Méthodes pour la génération des traces d'exécution

Une variété d'agents pour la génération de traces d'exécution pour les applications mobiles ont été développés pour améliorer le processus de test. Les agents basés sur l'aléatoire comme MonkeyRunner (Developers, 2010) utilisent des séquences pseudo-aléatoires pour simuler les interactions utilisateur, offrant une exploration large, bien que non ciblée, de l'application sous le test (AUT). D'autre part, les agents basés sur des modèles tels que Droidbot (Li *et al.*, 2017) et son itération avancée, DroidBotX (Yasin, Hamid & Raja Yusof, 2021), emploient une approche structurée du test, utilisant des modèles pré-définis pour guider la génération des entrées, offrant ainsi une couverture de test plus ciblée et efficace, mais ne prenant pas en compte le contexte de l'application, ce qui peut parfois entraîner un échec à passer l'activité de connexion. HumanDroid (Li, Yang, Guo & Chen, 2019) étend ce concept en intégrant des interactions similaires à celles des humains, visant à imiter plus étroitement le comportement des utilisateurs réels en utilisant un réseau de neurones profond. Cependant, l'entraînement de tels modèles est coûteux en temps et nécessite une grande quantité de données.

L'émergence d'agents basés sur LLM marque une évolution significative dans ce domaine. Des outils comme GPTDroid (Liu *et al.*, 2023) et Droidbot-GPT (Wen *et al.*, 2024) exploitent les capacités sophistiquées des LLM pour générer des interactions *sémantiquement* riches et conscientes du contexte avec l'AUT. DroidAgent (Yoon *et al.*, 2024), en particulier, représente un bond en avant, en fixant et en poursuivant de manière autonome des objectifs de tâches au sein de l'environnement de l'application. Sa capacité à naviguer à travers plus d'activités et à effectuer des tâches significatives spécifiques à l'application souligne son potentiel en tant

qu'outil puissant dans les tests d'interface graphique. Cependant, le coût-efficacité de tels agents avancés, comme en témoignent les coûts opérationnels associés à DroidAgent, reste un facteur critique pour une adoption généralisée dans l'industrie. Ce travail développe une version de l'outil DroidAgent qui utilise des LLM librement accessibles.

1.5 L'analyse dynamique pour la détection des défauts de code

Dynamics (Prestat *et al.*, 2024) est la première méthode outillée permettant de détecter sept types de défauts de code comportementaux spécifiques à Android à l'aide de l'analyse dynamique. Cependant, son rappel n'est que de 53,4%, en raison d'un nombre élevé de faux négatifs, principalement attribuable à la faible couverture des outils de génération de traces d'exécution basés sur des stratégies aléatoires, tels que MonkeyRunner. DynamicsLLM est également basé sur la méthode Dynamics, et pour résoudre ce défi, un générateur de traces basé sur LLM est utilisé pour produire des traces d'exécution intelligentes. Il y a un intérêt considérable pour la détection des défauts de code et la réalisation d'une analyse dynamique des applications mobiles, soulignant la concentration de la communauté sur ces sujets. Cependant, il existe une lacune notable dans la recherche concernant la détection des défauts de code comportementaux par analyse dynamique dans les applications mobiles, ainsi qu'un manque d'outils et de méthodes dédiés à cette fin. Cette lacune souligne la pertinence de ce travail dans l'avancement de l'état de l'art.

1.6 Les grands modèles de langage (LLM) pour la détection des défauts de code

Des études récentes ont exploré l'application des LLM pour détecter et corriger les défauts de code. Divers LLM sont utilisés à cette fin, y compris des modèles propriétaires comme la série GPT-4 (Mesbah, El Madhoun, Al Agha & Chalouati, 2025) et Gemini (Amorim, da Costa, Alves & Figueiredo, 2025), ainsi que des alternatives librement accessibles comme LLaMa (Mesbah *et al.*, 2025) et DeepSeek (Sadik & Govind, 2025). Le cadre iSmell (Wu *et al.*, 2024) utilise une approche Mixture of Experts (MoE), intégrant plusieurs outils de détection de défauts de code pour fournir une analyse complète et affinant les LLM avec les résultats des outils experts

pour faciliter une refactorisation efficace. Cependant, les méthodologies existantes reposent principalement sur l'analyse statique, qui est insuffisante pour identifier les défauts de code comportementaux et nécessite l'accès au code source.

Ce travail se distingue par l'utilisation de LLM librement accessibles dans le cadre de l'analyse dynamique afin de générer des traces d'exécution intelligentes. Cette approche permet de pallier ces limitations et d'améliorer le processus de détection sans nécessiter l'accès au code source. De plus, deux approches *hybrides* sont proposées : l'une basée sur une heuristique et l'autre sur l'algorithme UCB, afin d'accroître la couverture.

1.7 Conclusion

De nombreuses recherches ont porté sur la détection des défauts de code dans les applications mobiles par l'analyse dynamique, ainsi que sur diverses techniques de génération de traces d'exécution, ce qui démontre l'intérêt de la communauté scientifique pour ces sujets. En revanche, à ce jour, aucun travail n'a exploité d'outils de génération de traces d'exécution basés sur les LLM dans le cadre de l'analyse dynamique pour la détection des défauts de code. Il existe donc un manque d'évaluations concernant leurs capacités à générer des traces d'exécution intelligentes, ainsi que leurs performances et leurs limites. Ceci met en évidence l'intérêt et la pertinence des contributions de ce travail à l'état de l'art.

CHAPITRE 2

CONCEPTS PRÉLIMINAIRES

Ce chapitre présente les concepts préliminaires sur lesquels s'appuie ce travail. D'abord, les défauts de code comportementaux sont définis avec leurs comportements. Ensuite, présenter l'outil DroidAgent utilisé dans l'implémentation de DynamicsLLM.

2.1 Les défauts de code comportementaux

Seuls 7 *défauts de code comportementaux* Android ont été identifiés dans la littérature par (Prestat *et al.*, 2022). Les définitions fournies dans (Hecht *et al.*, 2015; Prestat *et al.*, 2024) sont présentées, ainsi que les comportements inappropriés correspondants.

Durable WakeLock (DW) : Un *WakeLock* est un mécanisme qui permet à une application de maintenir l'appareil éveillé afin d'achever une tâche.

Comportement inapproprié : Un appel à la méthode *acquire* n'est pas suivi d'un appel à la méthode *release*. Cela provoque une fuite d'énergie.

Init OnDraw (IOD) : Les routines *onDraw* sont responsables de la mise à jour de l'interface graphique des applications Android. Ces routines sont invoquées à chaque rafraîchissement de l'interface graphique (jusqu'à 60 fois par seconde).

Comportement inapproprié : Tout calcul supplémentaire ou toute utilisation excessive de la mémoire effectuée dans *onDraw* est amplifié en raison de la fréquence élevée des invocations.

Heavy Processes (HP) : Cette catégorie peut être divisée en trois *défauts de code* : (1) *Heavy AsyncTask (HAS)*, où *AsyncTask* est utilisé pour exécuter des opérations lourdes en arrière-plan ; (2) *Heavy Service Start (HSS)*, où les services Android sont utilisés pour exécuter des opérations

lourdes ; et (3) *Heavy Broadcast Receiver (HBR)*, où un récepteur de diffusion est utilisé pour gérer la communication avec le système ou d'autres applications.

Comportement inapproprié : L'utilisation de *AsyncTask*, des services ou des *Broadcast Receivers* nécessite l'invocation de plusieurs méthodes de démarrage et de rappel. Certaines de ces méthodes sont exécutées sur le processus principal de l'interface utilisateur et ne doivent pas être coûteuses en temps ou bloquantes, car cela peut conduire le système à tuer l'application.

No Low Memory Resolver (NLMR) : La méthode *onLowMemory* est responsable de la réduction de l'utilisation de la mémoire d'une activité en cours d'exécution.

Comportement inapproprié : Si la méthode n'est pas implémentée ou ne parvient pas à libérer de la mémoire lors de son exécution, le système Android peut automatiquement tuer le processus de l'activité afin de libérer de la mémoire, entraînant potentiellement une terminaison inattendue du programme.

HashMap Usage (HMU) : La plateforme Android fournit *ArrayMap* et *SimpleArrayMap* comme remplacements plus économes en mémoire de *HashMap*. Ces alternatives déclenchent moins de ramasse-miettes sans différence significative de performance pour des structures contenant jusqu'à plusieurs centaines d'entrées.

Comportement inapproprié : Un *HashMap* est utilisé pour un petit ensemble d'objets, alors que *ArrayMap* et *SimpleArrayMap* sont utilisés pour de grands ensembles.

2.2 L'outil DroidAgent

DroidAgent est un générateur de traces basé sur les LLM, conçu selon une architecture orientée agents composée de quatre agents principaux basés sur des LLM : Planner, Actor, Observer et Reflector. Les agents Actor et Observer forment une boucle « interne », travaillant à l'accomplissement des tâches planifiées par le Planner et ensuite analysées par le Reflector, comme illustré à la figure 2.1. Le processus est décrit comme suit :

1. **Planification :** DroidAgent planifie en continu des tâches de haut niveau à effectuer. Ces tâches correspondent à des étapes sémantiquement significatives et s'alignent avec les fonctionnalités cohérentes de l'application cible. L'agent Planner génère des tâches viables et diversifiées tout en évitant la répétition de tâches impossibles, non pertinentes ou déjà accomplies grâce aux éléments suivants :

   - **Historique des tâches de haut niveau :** Il s'agit d'un résumé textuel des 20 tâches les plus récentes et des 5 éléments de connaissance les plus pertinents relatifs aux tâches. Ces informations historiques sont stockées en mémoire à long terme par l'agent Reflector.
   - **Activités totales et visitées :** Une liste des activités couvertes et non couvertes ainsi que le nombre de visites pour chaque activité, ce qui permet d'évaluer la progression de l'exploration et de fournir des détails sur toutes les activités, y compris leur état actuel.
   - **Connaissances initiales :** DroidAgent est initialisé avec des connaissances initiales telles que les informations du profil utilisateur virtuel (*Persona*).

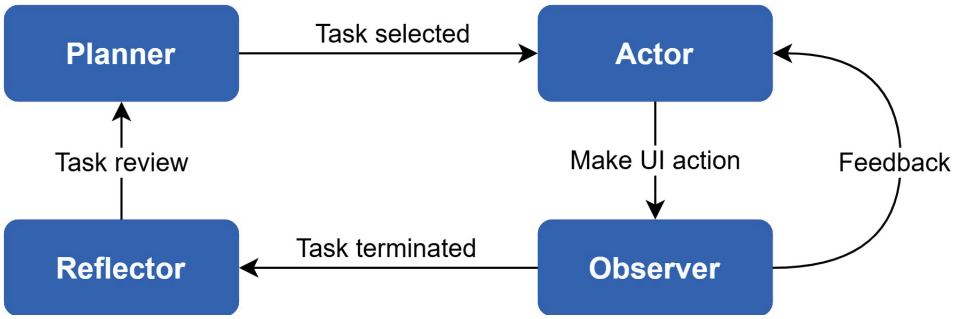

Figure 2.1 Processus de DroidAgent, tirée de Cherief *et al.* (2026a)

2. **Action :** DroidAgent sélectionne une tâche et fournit une invite (prompt) contenant les propriétés des éléments d'interface graphique (telles que le nom, le texte, le nombre de visites, la possibilité de cliquer, d'éditer, etc.) à l'agent Actor (LLM). L'agent LLM est ensuite chargé de choisir un élément d'interface graphique et de déterminer l'action appropriée (par exemple, cliquer). Si l'action sélectionnée nécessite un ou plusieurs paramètres, comme le texte à saisir pour une action d'édition, le LLM est invité à fournir les paramètres nécessaires en s'appuyant sur l'historique des invites et des réponses précédentes liées à l'action en cours. En cas d'erreur (par exemple, sélection d'une option incompatible avec l'élément d'interface graphique ou fourniture de paramètres incorrects, comme un type non valide pour un courriel ou un âge), une nouvelle invite est envoyée. Cette invite inclut l'historique des invites-réponses de la tâche en cours ainsi que le message d'erreur. Après chaque trois itérations, l'agent fournit un retour basé sur l'historique des invites et réponses de la tâche, ainsi que sur les observations. Ce retour aide à prévenir des actions indésirables (par exemple, quitter l'application) ou la répétition de la même action, guidant l'Actor dans la détermination de l'étape suivante afin de mener à bien la tâche en cours.
3. **Observation :** Après chaque action effectuée, DroidAgent met à jour sa perception de l'écran et envoie une invite à l'agent Observer afin de comparer les états précédents et actuels de l'interface graphique. Ces informations fournissent un retour à l'agent Actor sur l'efficacité de l'action choisie. Ensuite, DroidAgent procède à l'étape d'action suivante, suivie d'une nouvelle observation. Ce cycle se poursuit jusqu'à ce que la tâche en cours soit terminée ou que l'agent Actor appelle la fonction « end task ».
4. **Réflexion sur la tâche :** L'agent Reflector est activé à la fin du cycle d'exécution d'une tâche, que celui-ci se termine par l'appel de la fonction « end task » par l'agent Actor ou par l'atteinte de la limite maximale d'actions. Reflector examine l'historique d'exécution de la tâche, y compris l'auto-critique, les observations, l'état actuel de l'interface graphique et l'objectif de la tâche, et génère un résumé concis incluant une étiquette binaire de succès ou d'échec. L'objectif du Reflector est de produire des réflexions qui informent la planification future des tâches et améliorent l'efficacité dans l'atteinte de l'objectif global. Ce processus de réflexion permet de préserver les connaissances utiles issues de l'exécution des tâches,

d'éviter que le système n'oublie des informations importantes et de réduire le risque que les autres agents LLM s'écartent de leur objectif initial ou génèrent des sorties non pertinentes. Une fois la réflexion terminée, le processus redémarre à l'étape 1 et se poursuit jusqu'à l'expiration du temps d'exécution.

Les principales stratégies d'ingénierie des invites utilisées sont :

- **Décomposition des tâches :** L'agent PLANNER décompose les tâches complexes en sous-tâches plus petites, ce qui permet de créer des invites plus précises pour le LLM.
- **Raisonnement par chaîne de pensée :** DROIDAGENT adopte une approche étape par étape pour raisonner à travers les tâches.
- **Auto-réflexion :** L'agent REFLECTOR améliore les décisions futures en se basant sur les expériences passées.
- **Utilisation des outils :** L'agent ACTOR exploite les actions de l'interface graphique et fournit les paramètres nécessaires grâce à cette technique.
- **Gestion de la mémoire :** DROIDAGENT utilise des systèmes de mémoire à court terme et à long terme durant l'exécution, afin de poursuivre et de construire sur les connaissances précédentes.
- **Paramètre de température :** Il détermine le niveau d'aléatoire dans les sorties du modèle. Des températures plus élevées conduisent à des réponses plus variées et diversifiées. Le réglage de la température à 0,6 permet d'atteindre un équilibre, produisant des sorties plus factuelles et fiables (OpenAI, 2025).

2.3 Conclusion

Ce chapitre introduit les concepts de base nécessaires à la compréhension de ce mémoire, à savoir les défauts de code comportementaux et également l'architecture de l'outil DROIDAGENT et les principales stratégies d'ingénierie des invites utilisés. Le prochain chapitre expose l'outil DYNAMICSLLM. Les approches de génération de traces d'exécution utilisées sont également présentées notamment : l'approche basée uniquement sur les LLM, l'approche *hybride* heuristique et l'approche fondée sur l'algorithme UCB.

CHAPITRE 3

DYNAMICSLLM : OUTIL POUR LA DÉTECTION DES DÉFAUTS DE CODE COMPORTEMENTAUX

Ce chapitre présente dans la section 3.1 l'outil DynamicsLLM, une implémentation de la méthode Dynamics, en détaillant chacune de ses étapes. Ensuite, dans la section 3.2, les approches utilisées lors de l'étape d'exécution afin d'améliorer la couverture des événements liés aux défauts de code, qui constituent l'objectif de ce mémoire.

3.1 L'outil DynamicsLLM

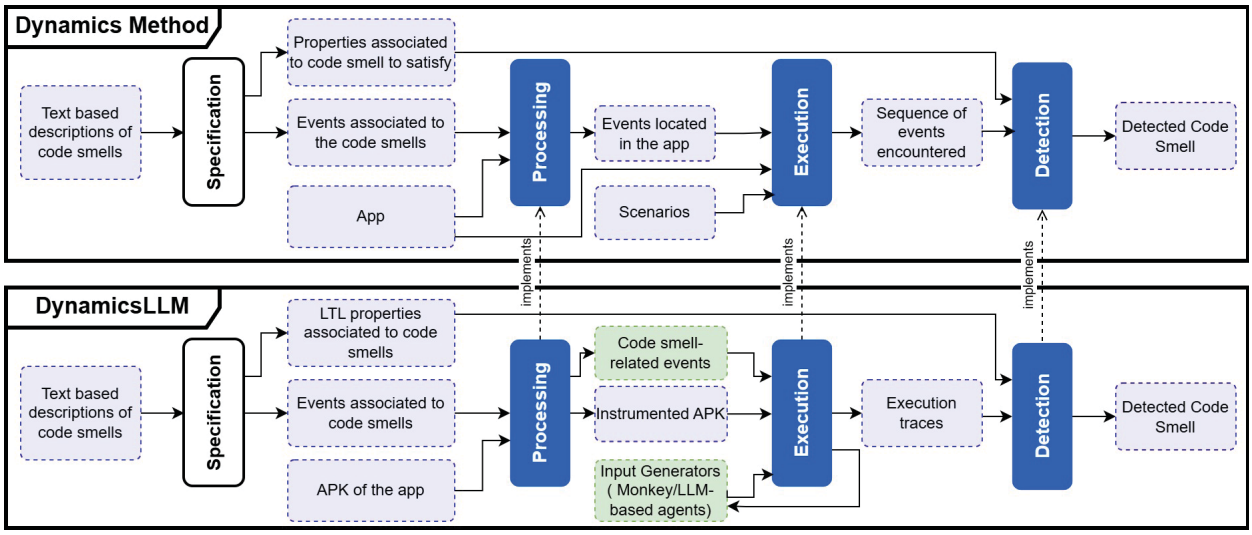

Figure 3.1 Les différentes étapes de la méthode Dynamics et de l'outil DynamicsLLM. Les boîtes représentent les étapes, et les flèches relient les entrées et les sorties de chaque étape, décrites par des boîtes en pointillés. Les boîtes en pointillés verts mettent en évidence les nouveaux éléments ou ceux modifiés par rapport à l'outil Dynamics, tirée de Cherief *et al.* (2026a)

L'outil DynamicsLLM est une implémentation concrète de la méthode Dynamics (Prestat *et al.*, 2024), qui généralise l'outil Dynamics en intégrant l'utilisation des LLM pour générer intelligemment des traces d'exécution. Les différentes étapes de la méthode Dynamics et de l'outil DynamicsLLM sont illustrées dans la figure 3.1. Les étapes *Specification* et *Detection* n'ont pas été modifiées ; seules les étapes *Processing* et *Execution* ont été modifiées. Chaque

étape de l'outil DynamicsLLM sera présentée ci-dessous, en détaillant explicitement les entrées, les sorties, la description, l'implémentation et un exemple d'exécution.

3.1.1 Étape 1. Spécification

Entrées : Descriptions textuelles des *défauts de code* mobiles identifiés dans la littérature.
Sorties : Les événements liés aux *défauts de code* et la propriété en logique temporelle linéaire (LTL) associée à chaque *défaut de code*, dont la satisfaction indique la présence du défaut.
Description : Les événements liés aux *défauts de code* sont extraits de leurs descriptions et utilisés pour définir les propriétés associées. Chaque événement est caractérisé par des éléments spécifiques du code source, tels que des appels de méthodes ou des structures de code, et est associé à des valeurs telles que des horodatages ou des métriques de mémoire.
Implémentation : Cette étape ne comporte pas d'implémentation. La spécification, incluant les définitions des événements et des propriétés, a été réalisée par Prestat *et al.* (2024).
Exemple d'exécution : Les *événements HMU* sont déclenchés par des appels aux méthodes des classes *HashMap*, *SimpleArrayMap* et *ArrayMap*, qui influencent la taille des structures : *new*, *put*, *putAll*, *remove*, *clear*. Les valeurs associées à chaque événement sont le type de structure, sa classe, sa taille réelle et son identifiant.

3.1.2 Étape 2. Traitement

Entrées : Le fichier APK de l'application et les événements liés aux *défauts de code*.
Sorties : L'APK est instrumenté avec des instructions de log, ainsi qu'un fichier contenant l'ensemble des événements spécifiques pouvant provoquer un *défaut de code*.
Description : L'objectif de cette étape est d'instrumenter l'APK d'une application mobile en ajoutant des instructions de log qui varient selon la catégorie du *défaut de code*. Cela implique d'identifier les événements pertinents dans l'APK et d'insérer des instructions générant des entrées de log spécifiques à chaque événement. Ainsi, lors de l'exécution de l'application instrumentée, une trace, constituée d'une séquence d'entrées de log, est produite. Dans le

contexte de ce travail, une entrée de log est représentée par un tuple (location, id, event, values), où :

- *location* correspond aux noms du paquet, de la classe et de la méthode où l'événement se produit ;
- *id* est un identifiant séquentiel permettant de distinguer plusieurs occurrences du même événement dans une même méthode d'une même classe ;
- *event* est le mot-clé associé à un événement lié à un *défaut de code* ;
- *values* contient les valeurs utilisées pour les propriétés LTL de détection du *défaut de code*, telles que la taille ou le temps.

Contrairement à Dynamics, DynamicsLLM génère également un autre fichier contenant des informations sur l'ensemble des événements liés aux *défauts de code*, structurées sous la forme d'un tuple (classe, méthode, numéro de ligne, événement).

Implémentation : Cette étape repose sur une analyse statique visant à instrumenter le code. Pour cela, un ensemble de fonctionnalités permettant d'enregistrer les événements dans un fichier après la fin de l'instrumentation a été ajouté au module d'implémentation de Dynamics.

L'outil DynamicsLLM, en s'appuyant sur Soot (Vallée-Rai *et al.*, 2010), traite la représentation de chaque classe, méthode et instruction, en les convertissant en Jimple (Vallée-Rai *et al.*, 2010), une version simplifiée du code source Java. Les instructions sont ensuite analysées afin de déterminer si des événements sont présents. Lorsqu'un événement est détecté, des instructions générant des entrées de log sont insérées immédiatement après l'événement dans l'APK. Enfin, l'APK généré doit être signé afin d'être exécuté lors de l'étape suivante, réalisé à l'aide de Apksigner (Corporation, 2025).

DynamicsLLM et Dynamics analysent le bytecode plutôt que le code source, ce qui présente des avantages et des inconvénients. D'un côté, l'accès au code source et les étapes de compilation ne sont pas nécessaires. En revanche, la traduction du bytecode vers Java ou d'autres langages intermédiaires est complexe en raison de pertes d'information importantes, telles que les numéros

```
HashMap<String, TimePeriod> CACHE = new HashMap<String, TimePeriod>();
CACHE.put(string, value = new TimePeriod(begin, end));
```

Figure 3.2 Exemple d'instruction Java,
tirée de Cherief *et al.* (2026a)

```
package.TimePeriodPreference.java$fromString:0:hmuadd:20398:1:HashMap
```

Figure 3.3 Sortie de log associée à l'instruction Java,
tirée de Cherief *et al.* (2026a)

de ligne originaux. De plus, le type et le nom des variables locales sont difficiles à récupérer, et les instructions *GOTO* remplacent les structures de contrôle (boucles et conditions).

Exemple d'exécution : La figure 3.2 montre un appel à la méthode *put* du *défaut de code HMU*, qui ajoute un élément à la structure **HashMap**. Il s'agit d'un événement d'*Addition* pour le *défaut de code HMU*. Dans ce cas, l'application instrumentée produira l'entrée de log illustrée à la figure 3.3. Cette entrée indique le nom de la méthode *fromString*, le nom de la classe *TimePeriodPreference* et le nom du paquet *package* où l'appel a lieu. Elle montre également qu'il s'agit du premier événement de ce type dans la méthode grâce à l'identifiant 0. Enfin, elle précise qu'il s'agit d'une *Addition* (mot-clé *hmuadd*), que l'opération a été effectuée sur la structure d'identifiant 20398, que la taille est de 1 et que le type de structure est **HashMap**.

3.1.3 Étape 3. Exécution

Entrées : L'APK instrumenté, le fichier des événements liés aux *défauts de code* et le générateur d'entrées basé sur les LLM.
Sorties : Les traces d'exécution obtenues après l'exécution de l'application instrumentée à l'aide d'un générateur de traces basé sur les LLM.
Description : Cette étape consiste à exécuter l'application instrumentée sur un appareil réel ou virtuel à l'aide d'un agent basé sur les LLM, configuré avec différents paramètres (fichier

APK, fichiers d'événements, temps d'exécution autorisé, identifiant du périphérique, répertoire de sortie) afin de produire des traces d'exécution. Cet agent est un programme qui simule les interactions utilisateur en générant des entrées pour l'application, telles que le clic sur un bouton, la saisie de texte dans un champ ou la navigation arrière. Cette étape est cruciale pour l'outil DynamicsLLM, car les traces générées facilitent la détection des *défauts de code*. Chaque fois que l'exécution atteint une instruction de log, une entrée correspondante est produite. Par conséquent, la trace d'exécution regroupe l'ensemble de ces entrées, reflétant les événements associés aux *défauts de code* détectés. Le principal défi consiste à obtenir une couverture aussi complète que possible du code instrumenté lors de l'exécution afin de capturer un maximum d'événements.

Implémentation : DroidAgent a été choisi en raison de ses capacités avancées basées sur les LLM. Les outils fondés sur les LLM offrent une intelligence accrue, nécessitent peu d'entraînement et sont conçus pour comprendre le contexte des applications, ce qui leur permet de planifier dynamiquement des séquences d'actions afin d'atteindre leurs objectifs. GPTDroid a été écarté car son code n'est pas accessible publiquement (Yoon *et al.*, 2024) et il fonctionne uniquement comme un agent acteur. En revanche, DroidAgent opère comme un agent orienté tâches, planifiant les étapes au cours de l'exécution en fonction des tâches assignées.

Les principales modifications apportées à DroidAgent sont présentées ci-dessous, avec des exemples d'invites mis à jour inclus dans l'annexe I.

- **Utilisation de LLM librement accessibles :** Tous les agents LLM reposent sur des modèles de langage librement accessibles. Le module de modèles de DroidAgent est mis à jour, afin d'envoyer des invites à des LLM exécutés localement ou sur un serveur via (Ollama, 2024), supprimant ainsi la dépendance exclusive au modèle GPT de la version originale.
- **Redéfinition de l'objectif final :** Pour l'agent Planner, le nouvel objectif a été modifié de « [PERSONA]'s ultimate goal is to visit as many pages as possible and try their core functionalities » à « [PERSONA]'s ultimate goal is to trigger code smell-related events as much as possible ».

- **Extension des connaissances initiales :** Les connaissances initiales de l'agent PLANNER sont enrichies en y intégrant une liste de *défauts de code* potentiels, obtenue lors de l'instrumentation et formatée comme suit : « (Type de défaut de code, classe, méthode) : [...]». Cette liste est également utilisée lors de l'étape d'observation pour identifier les événements déclenchés.
- **Invite d'observation :** À chaque étape d'observation, l'invite envoyée à l'agent OBSERVER inclut à la fois les changements visibles de l'activité et des informations détaillées sur les événements liés aux *défauts de code* déclenchés. Plus précisément, elle rapporte : (i) le nombre d'événements depuis la dernière observation, (ii) le nombre cumulé d'événements déclenchés, et (iii) le nombre total d'événements liés aux *défauts de code* présents dans l'application.

Cette solution est entièrement automatisée. L'outil DYNAMICSLLM est utilisé sur l'émulateur GOOGLE ANDROID EMULATOR (Android, 2024) afin de générer les traces qui seront utilisées pour la détection. L'outil en ligne de commande ADB (Android, 2025a) est utilisé pour installer l'APK, UIAUTOMATOR2 (He & openatx, 2024) pour extraire la hiérarchie des vues, et l'outil en ligne de commande LOGCAT (Android, 2025b) pour récupérer les journaux du périphérique, contenant les traces d'exécution générées. Ces journaux sont filtrés afin de ne conserver que ceux qui correspondent au format de journal. Ainsi, aucune filtration n'est appliquée aux séquences générées.

```
package.TimePeriodPreference.java$<clinit>:0:hmuimpl:20398:0:HashMap
package.TimePeriodPreference.java$fromString:0:hmuadd:20398:1:HashMap
package.TimePeriodPreference.java$fromString:0:hmuadd:20398:2:HashMap
package.TimePeriodPreference.java$fromString:0:hmuadd:20398:3:HashMap
package.TimePeriodPreference.java$fromString:0:hmuadd:20398:4:HashMap
```

Figure 3.4 Extrait d'une trace d'exécution, tirée de Cherief *et al.* (2026a)

Exemple d'exécution : La figure 3.4 présente un extrait d'une **trace générée** lors de l'exécution visant à détecter le **défaut de code HMU**. Cet extrait est issu de l'exemple d'exécution présenté à l'étape 2. La trace montre une **implémentation** et quatre instances d'ajout de l'identité 20398 dans la structure **HashMap**. Elle indique que la taille maximale atteinte est de 4 et que, tout au long de l'exécution, la **HashMap** a été utilisée pour un petit ensemble, ce qui constitue un *défaut de code HMU*.

3.1.4 Étape 4. Détection

Comme dans l'outil Dynamics, DynamicsLLM détecte les *défauts de code* à partir de la trace d'exécution complète de l'application.

Entrées : Un ensemble de traces d'exécution et les propriétés LTL associées aux *défauts de code*.

Sorties : Les *défauts de code* détectés durant l'exécution sont regroupés dans des fichiers par type.

Description : Cette étape consiste à analyser les traces d'exécution afin d'identifier les *défauts de code* survenus, en vérifiant les séquences d'événements dans les traces et la satisfaction des propriétés associées. Seuls les événements liés à une instance ou à un appel de méthode spécifique sont pris en compte pour la vérification, afin d'éviter d'utiliser des événements provenant d'objets ou d'appels de méthode différents.

Implémentation : Cette étape utilise un module Java basé sur la plateforme d'analyse statique SOOT. En alternative, BeepBeep 3 (Hallé & Khoury, 2017), intégré à l'outil Dynamics, peut être utilisé pour le traitement des traces d'exécution et des flux d'événements. Dans les deux cas, une branche est spécifiée pour chaque *défaut de code* afin de vérifier la propriété correspondante. Une fois la trace entièrement analysée, les *défauts de code* détectés sont identifiés. Chaque *défaut de code* est ajouté au fichier correspondant sous forme de tuple contenant toutes les informations nécessaires ; la structure de ce tuple varie selon le type de *défaut de code*.

07:52:02.035,package.TimePeriodPreference.java$<clinit>,0,**hmuimpl**

Figure 3.5 Format d'un défaut détecté,
tirée de Cherief *et al.* (2026a)

Exemple d'exécution : Pour le *défaut de code* détecté à la figure 3.4, une nouvelle ligne est ajoutée au fichier des événements HMU, comme illustré à la figure 3.5. Les entrées incluent le nom de l'APK, le paquet, le fichier contenant la classe et la méthode (ici, **clinit**, le constructeur). Les détails inclus dépendent du type de *défaut de code*. Dans le cas de **HashMap**, deux informations supplémentaires sont fournies : le *type de structure* (HashMap, ArrayMap ou SimpleArrayMap) et la *taille maximale* atteinte durant l'exécution, qui est de 4.

3.2 Approches de génération des traces d'exécution

Cette section décrit les différentes approches utilisées pour la génération de traces d'exécution lors de l'étape d'exécution.

3.2.1 Approche aléatoire

Cette approche consiste à utiliser l'outil MonkeyRunner, un outil intégré dans la SDK Android, qui génère des flux pseudo-aléatoires d'événements utilisateur tels que des clics, des appuis ou des gestes, ainsi que plusieurs événements au niveau du système (Developers, 2010).

3.2.2 Approche 100% LLM

Cette approche consiste à utiliser DroidAgent uniquement avec les modifications décrites dans la section précédente.

3.2.3 Approche hybride heuristique

Ce travail présente une nouvelle méthode pour l'étape d'*Exécution*, combinant une approche basée sur les LLM et une approche aléatoire. Plutôt que de s'appuyer sur des méthodes classiques reposant uniquement sur le temps, MonkeyRunner est utilisé pour explorer les applications mobiles. Lorsque DynamicsLLM détecte que MonkeyRunner est « bloqué » (c'est-à-dire qu'il ne permet plus d'identifier de nouveaux événements liés aux *défauts de code* au cours des cinq dernières minutes), il bascule vers DroidAgent au cours de cinq étapes afin de « débloquer » MonkeyRunner. Ensuite, MonkeyRunner est réutilisé jusqu'à ce qu'il soit de nouveau bloqué.

L'agent LLM (DroidAgent) produit des actions de haute qualité, mais fonctionne lentement, tandis que l'agent aléatoire (MonkeyRunner) est très rapide, mais manque d'intelligence. Cette méthode exploite ainsi les forces des deux approches afin de maximiser la couverture.

Pour déterminer le *nombre de minutes* nécessaire pour détecter le blocage et le *nombre d'actions* requis pour le débloquer, ces hyperparamètres sont variés sur un échantillon de 15 applications mobiles. Les meilleures performances ont été obtenues avec un seuil de 5 minutes et de 5 actions.

3.2.4 Approche hybride basée sur UCB

Cette approche repose sur l'algorithme d'apprentissage par renforcement appelé *Upper Confidence Bound* (UCB), utilisé pour optimiser le compromis exploration–exploitation. À chaque prise de décision, deux stratégies sont envisagées : utiliser l'approche aléatoire pendant un nombre fixe d'actions, ou utiliser le LLM pendant une durée limitée. L'objectif est de sélectionner la stratégie la plus prometteuse en fonction de l'estimation de sa valeur attendue et de son incertitude.

Concrètement, UCB attribue à chaque stratégie un score combinant la performance observée jusqu'à présent (exploitation) et un terme favorisant les stratégies moins explorées (exploration). La stratégie ayant le score le plus élevé est alors sélectionnée, ce qui permet d'explorer de nouvelles actions tout en exploitant celles qui se sont révélées efficaces. Dans ce cas, ce

mécanisme permet d'adapter dynamiquement la stratégie employée en fonction de chaque application mobile et de l'évolution de l'exploration.

La stratégie LLM est évaluée en fonction du temps et la stratégie aléatoire en fonction du nombre d'actions. En effet, si c'est fixé aux deux stratégies un nombre d'actions très faible (par exemple une ou quelques actions), l'UCB sélectionnera presque toujours la stratégie LLM, car la majorité des actions aléatoires sont peu efficaces. À l'inverse, si un nombre d'actions très élevé est fixé, le LLM étant plus lent, il offrira à l'UCB moins d'occasions de prendre une décision, ce qui limite son adaptation.

La fixation des deux stratégies en fonction du temps n'est pas souhaitable, car MonkeyRunner peut exécuter un nombre très variable d'actions dans un même intervalle temporel, selon les performances de l'émulateur et de l'application mobile.

Pour déterminer le nombre de minutes à allouer au LLM et le nombre d'actions à attribuer à MonkeyRunner, une variation de ces hyperparamètres a été faite sur un échantillon de 40 applications mobiles sélectionnées aléatoirement. Les meilleures performances ont été obtenues avec deux configurations : (i) 2 minutes pour le LLM et 15 actions aléatoires, et (ii) 2 minutes pour le LLM et 60 actions aléatoires.

3.3 Conclusion

Dans ce chapitre, l'outil DynamicsLLM l'implémentation de la méthode Dynamics est présenté, en détaillant chacune de ses étapes. Ensuite, les approches, utilisées lors de l'étape d'exécution pour améliorer la couverture des événements liés aux défauts de code, sont exposées. Dans le chapitre 4 suivant, la méthodologie d'évaluation, les résultats obtenus et les limitations identifiées seront présentés.

CHAPITRE 4

VALIDATION ET EXPÉRIMENTATIONS

Ce chapitre présente une analyse de DynamicsLLM qui s'appuie sur les différentes approches décrites dans le chapitre précédent, en le comparant à l'outil Dynamics, considéré comme l'état de l'art. Cette étude comparative vise à fournir des indications précieuses sur l'efficacité, les capacités et les limites de chaque approche, en s'appuyant sur plusieurs métriques telles que la précision, le rappel, le nombre de défauts de code détectés et la couverture des événements liés à ces défauts. Elle permet également de guider les futures améliorations et développements.

4.1 Questions de recherche

Les questions de recherche abordées sont les suivantes :

- QR_1 **: Est-ce que les traces intelligentes générées par DynamicsLLM permettent de détecter les défauts de code comportementaux ?** Cette question étudie l'efficacité de DynamicsLLM pour la détection des défauts de code comportementaux en termes de précision et de rappel.
- QR_2 **: Est-ce que DynamicsLLM couvre davantage d'événements liés aux défauts de code que l'outil de l'état de l'art ?** Cette question analyse la couverture de DynamicsLLM en ce qui concerne le déclenchement des événements liés aux défauts de code.
- QR_3 **: Quelle est la contribution des LLM à l'efficacité globale ?** Cette question de recherche vise à déterminer s'il est toujours pertinent d'utiliser Dynamics et dans quelles conditions ou scénarios DynamicsLLM offre des avantages significatifs.
- QR_4 **: Est-ce que l'approche UCB permet d'améliorer la couverture des événements liés aux défauts de code par rapport à l'approche *hybride heuristique* ?** Cette question analyse la couverture de DynamicsLLM en utilisant les approches *hybrides*, notamment en ce qui concerne le déclenchement des événements liés aux défauts de code.

4.2 Configuration expérimentale

Le Tableau 4.1 présente l'ensemble des configurations utilisées pour répondre aux questions de recherche. Il est à noter que les approches *hybrides* sont configurées par DroidAgent pour exécuter trois actions aléatoires via MonkeyRunner toutes les trois secondes, afin d'éviter les problèmes de déconnexion de l'émulateur.

Pour l'approche LLM, l'exécution a été lancée en visant 100 actions, avec un budget maximal de trois heures. Toutefois, si l'exécution des 100 actions se termine en moins d'une heure, l'exécution est poursuivie jusqu'à atteindre une durée minimale d'une heure.

L'objectif fixé est de 100 actions parce que les performances d'un LLM dépendent fortement de la machine sur laquelle il est exécuté. De plus, avec les avancées industrielles et scientifiques constantes dans ce domaine, les modèles continueront de s'améliorer : ils deviendront plus légers et plus performants dans le futur.

Tableau 4.1 Les configurations d'approches utilisées

Approche	**Configuration**	**Désignation**	**Condition d'arrêt**
Aléatoire	MonkeyRunner	MonkeyRunner	1 heure
LLM	DroidAgent	100% LLM	100 actions (1–3 h)
Heuristique	5 minutes, 5 actions	Hybride	1 heure
UCB	2 minutes LLM, 15 actions aléatoires	UCB 02-15	1 heure
	2 minutes LLM, 60 actions aléatoires	UCB 02-60	1 heure

Comparativement au jeu de données utilisé par Prestat *et al.* (2024), les applications qui ne pouvaient pas être instrumentées ou exécutées avec la version sélectionnée du SDK Android sont exclues. Un sous-ensemble de 307 applications mobiles a été sélectionné à partir du jeu de données libre d'accès provenant de F-Droid (Community, 2025). Parmi celles-ci, seules 136 applications ont été annotées manuellement par Prestat *et al.* (2024) à des fins de validation,

représentant un échantillon stratifié statistiquement significatif avec un niveau de confiance de 95% et une marge d'erreur de 10%.

Les métriques collectées sont les suivantes :

- **Précision :** la proportion de vrais positifs (TP) parmi l'ensemble des positifs détectés (D) (c.-à-d. les défauts de code détectés), calculée comme $\frac{|TP|}{|D|}$.
- **Rappel :** la proportion de vrais positifs (TP) parmi la somme des vrais positifs et des faux négatifs (FN), calculée comme $\frac{|TP|}{|TP \cup FN|}$.
- **Nombre d'événements liés aux défauts de code couverts :** Le nombre de lignes instrumentées dans le code atteint durant la génération des traces, comptabilisées une seule fois par ligne.

4.3 Environnement d'expérimentation

Toutes les expériences ont été menées sur une machine Ubuntu 22.04 64 bits en utilisant l'émulateur Android de Google (Android, 2024) (API 35), ainsi que les Android SDK Command Line Tools 16.0 (Android, 2025b). Cette version a été choisie car DroidAgent ne parvenait pas à remplir les champs de texte dans les versions précédentes, en raison d'une incompatibilité avec la version du Droidbot. La configuration est connectée au serveur API RESTful (Ollama, 2024), qui gère les modèles LLM exécutés sur un serveur GPU.

mistral-small3.1 24b a été sélectionné sur la base d'une analyse préliminaire de 15 applications Android représentatives. Les résultats, illustrés dans la figure 4.1, montrent le nombre d'événements liés aux défauts de code déclenchés par notre implémentation DroidAgent en fonction du nombre d'actions effectuées. Notamment, **GPT-4o** a atteint le nombre d'événements le plus élevé avec un nombre réduit d'actions, indiquant une meilleure efficacité de couverture. **mistral-small3.1 24b** a également démontré des performances compétitives, mettant en évidence sa capacité à effectuer des actions pertinentes dès les premières étapes de l'exécution.

La figure 4.2 montre le nombre d'événements liés aux défauts de code déclenchés par DroidAgent en fonction du temps par chaque modèle LLM. **GPT-4o** a déclenché 268 évènements en seulement 34 minutes. **mistral-small3.1 24b** dépasse largement les modèles LLM librement accessibles

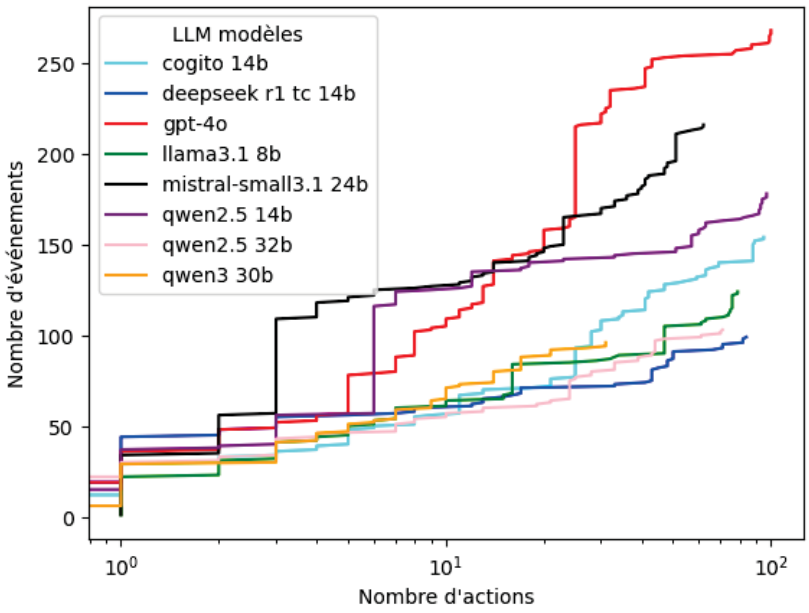

Figure 4.1 Nombre d'événements liés aux défauts de code couverts par chaque LLM, adaptée de Cherief *et al.* (2026a)

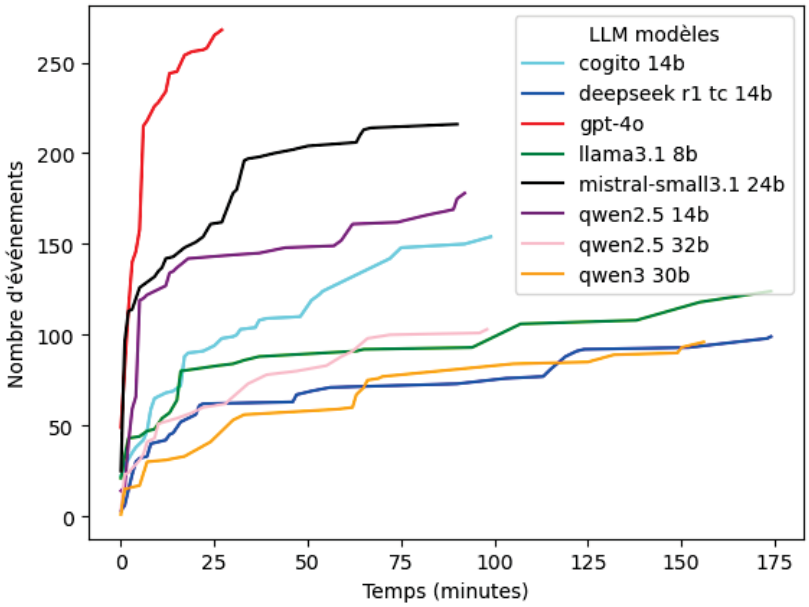

Figure 4.2 Nombre d'événements liés aux défauts de code couverts par chaque LLM

choisis, les plus légers que lui comme **llama 3.1 8b** et **qwen2.5 14b**, comme les plus grands **qwen3 30b** et **qwen2.5 32b**. Tous ces modèles ont réussi à passer LoginApp, à l'exception de **llama 3.1 8b**, probablement parce qu'il ne s'agit pas d'un modèle de raisonnement. Le modèle **mistral-small3.1 24b** a été choisi car l'objectif était d'utiliser un modèle librement accessible. Ainsi, **GPT-4o** est inclus uniquement comme référence de comparaison.

4.4 Résultats et discussions

Dans cette section, chaque question de recherche est traitée séparément.

4.4.1 QR_1 : Est-ce que les traces intelligentes générées par DynamicsLLM permettent de détecter les défauts de code comportementaux ?

Pour répondre à cette question, l'efficacité de l'ensemble des outils a été évaluée en utilisant la précision et le rappel comme métriques.

Tableau 4.2 La précision de la détection des défauts de code de DynamicsLLM

Défaut de code	Hybride	100% LLM
DW	1 / 1 (100%)	2 / 2 (100%)
HP	8 / 8 (100%)	8 / 8 (100%)
IOD	2 / 2 (100%)	5 / 5 (100%)
NLMR	1033 / 1033 (100%)	1033 / 1033 (100%)
HMU	154 / 154 (100%)	156 / 156 (100%)

Le Tableau 4.2 présente la précision du DynamicsLLM sur l'ensemble des applications testées, différenciées selon les différents types de défauts de code. L'outil DynamicsLLM, utilisant les configurations 100% LLM et Hybride, a démontré une précision de 100% pour tous les types de défauts de code, ce qui signifie que toutes les anomalies détectées étaient **correctes**.

Le Tableau 4.3 présente les métriques de rappel pour DynamicsLLM sur l'ensemble des applications testées, réparties selon les types de défauts de code. La détection des défauts DW et IOD s'est révélée plus modeste. Pour DW, cela s'explique par la faible couverture de ce

Tableau 4.3 Rappel pour la détection des défauts de code tiré de Cherief *et al.* (2026a)

Défaut de code	Hybride	100% LLM
DW	1 / 8 (12%)	2 / 8 (25%)
HP	8 / 13 (61%)	8 / 13 (61%)
IOD	2 / 8 (25%)	5 / 8 (62%)
NLMR	1033 / 1057 (98%)	1033 / 1057 (98%)
HMU	154 / 211 (73%)	156 / 211 (73%)

type d'instructions, tandis que pour IOD, la performance plus limitée s'explique par le fait que DynamicsLLM ne génère pas d'interactions soutenues et rapides sur l'ensemble de la session, ce qui conduit à des schémas d'utilisation sensiblement différents de ceux des utilisateurs humains. En revanche, pour les défauts HMU et NLMR, toutes les configurations ont atteint un rappel élevé, avec 73% pour HMU et 98% pour NLMR.

> ***RQ*$_1$** : DynamicsLLM, utilisant à la fois les configurations 100% LLM et Hybride, est capable de détecter des défauts de code comportementaux spécifiques à Android, comme le démontrent sa précision (100%) pour tous les types de défauts de code et ses scores de rappel compétitifs.

4.4.2 QR_2 : Est-ce que DynamicsLLM couvre davantage d'événements liés aux défauts de code que l'outil de l'état de l'art ?

Afin d'évaluer le potentiel des approches basées sur les LLM, une comparaison des événements liés aux défauts de code couverts par chaque outil a été réalisée. La significativité des résultats a été évaluée à l'aide du test de McNemar, et les résultats détaillés sont fournis dans l'annexe II.

La figure 4.3 illustre le nombre d'événements liés aux défauts de code déclenchés au fil du temps par différentes approches, sur trois catégories d'applications : 192 applications comportant au plus 4 activités, 74 applications comportant entre 5 et 10 activités, et 41 applications comportant plus de 10 activités.

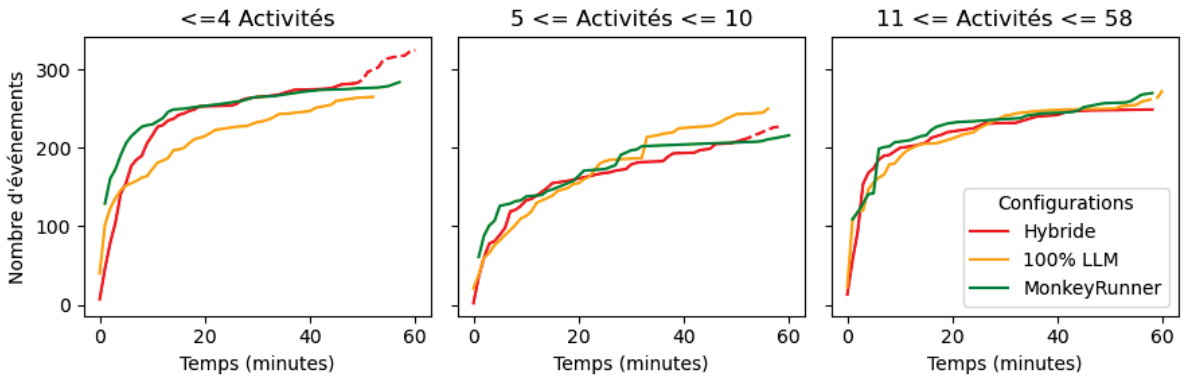

Figure 4.3 Événements liés aux défauts de code couverts par les configurations 100% LLM, HYBRIDE et MONKEYRUNNER au fil du temps. Les lignes pointillées indiquent que certaines exécutions se sont terminées prématurément, adaptée de Cherief *et al.* (2026a)

Dans une limite de temps d'une heure, la configuration HYBRIDE surpasse les autres pour les applications comportant peu d'activités, démontrant une couverture rapide et atteignant des améliorations de 14.4% et 22.6% par rapport à MONKEYRUNNER et 100% LLM, respectivement. Cette catégorie représente 63% du jeu de données. Les résultats du test de McNemar (Tableau II-1) confirment que les différences observées entre HYBRIDE et les autres configurations sont statistiquement significatives. Ces résultats suggèrent que la combinaison de stratégies est particulièrement efficace.

Pour les applications contenant entre 5 et 10 activités, 100% LLM atteint une couverture supérieure de 10.1% et 15.7% par rapport à HYBRIDE et MONKEYRUNNER, respectivement. Ces résultats mettent en évidence l'efficacité des approches basées sur les LLM pour cette catégorie ; les raisons sous-jacentes seront discutées plus en détail dans la réponse à la *RQ3*.

Les résultats du test de McNemar indiquent également des différences significatives entre la couverture obtenue par 100% LLM et celle des autres approches pour les applications comportant entre 5 et 10 activités, ainsi que pour celles comportant plus de 10 activités, pour lesquelles les performances finales sont globalement comparables entre les trois configurations.

Sur l'ensemble des catégories d'applications, 100% LLM présente initialement une performance inférieure à celle de MonkeyRunner durant les premières minutes, mais réduit progressivement cet écart au fil du temps. Cette latence initiale s'explique par le fait que chaque action proposée nécessite quatre étapes, chacune impliquant une ou plusieurs interactions avec le LLM, ce qui entraîne un coût temporel plus élevé.

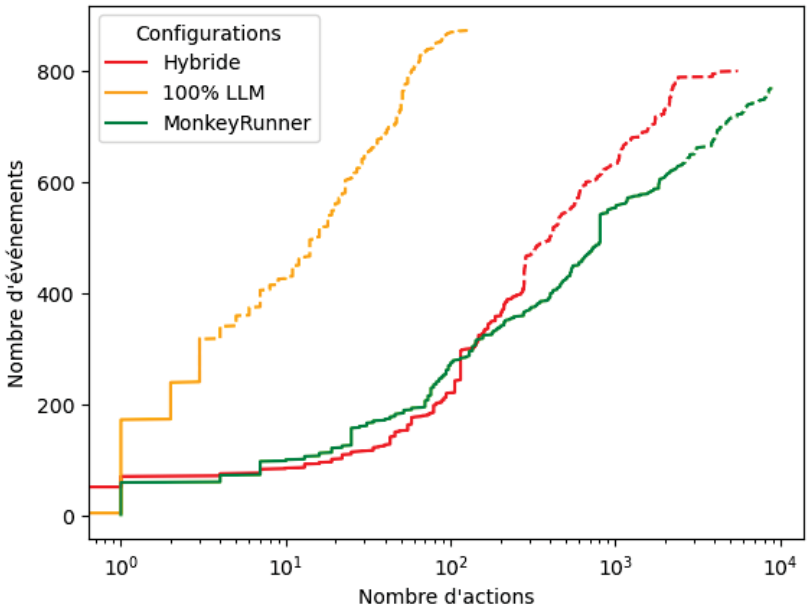

Figure 4.4 Nombre d'événements liés aux défauts de code couverts par les configurations 100% LLM, Hybride et MonkeyRunner en fonction du nombre d'actions. Les lignes pointillées indiquent que le délai d'exécution maximal a été atteint pour certaines exécutions, adaptée de Cherief *et al.* (2026a)

La figure 4.4 présente le nombre d'événements liés aux défauts de code déclenchés par chaque outil en fonction du nombre d'actions. Cette métrique est particulièrement pertinente au regard des avancées rapides des LLM. En outre, certaines applications peuvent être davantage contraintes par le nombre d'actions nécessaires pour compléter un test que par le temps d'exécution. À mesure que le nombre d'actions augmente, la complexité du débogage s'accroît également, rendant les échecs plus difficiles à analyser, reproduire et corriger.

La configuration 100% LLM démontre une excellente efficacité, atteignant 874 événements avec seulement 100 actions, soit près de trois fois plus que HYBRIDE et MONKEYRUNNER. Ce résultat confirme l'efficacité des LLM, même sans hybridation.

La configuration HYBRIDE se distingue également en découvrant le plus grand nombre d'événements tout en nécessitant significativement moins d'actions que MONKEYRUNNER. La similarité des résultats observés aux premières étapes entre HYBRIDE et MONKEYRUNNER s'explique par le fait que HYBRIDE s'appuie initialement sur MONKEYRUNNER durant les cinq premières minutes. Ces résultats montrent que DYNAMICSLLM ne se contente pas de découvrir davantage d'événements, mais qu'il le fait avec un nombre d'actions nettement inférieur. Cette efficacité s'explique par la capacité de l'exploration basée sur les LLM à raisonner sur l'interface utilisateur et à sélectionner des actions plus pertinentes, réduisant ainsi la redondance.

> ***QR*$_2$** : DYNAMICSLLM, utilisant la configuration 100% LLM, couvre davantage d'événements liés aux défauts de code pour un nombre d'actions donné, tandis que la configuration HYBRIDE est plus performante dans une contrainte de temps fixe pour les applications mobiles comportant peu d'activités.

4.4.3 QR_3 : Quelle est la contribution des LLM à l'efficacité globale ?

Cette question, subdivisée en sous-questions de recherche spécifiques, a été examinée à l'aide d'analyses à la fois quantitatives et qualitatives.

Tout d'abord, **les événements liés aux défauts de code couverts par DYNAMICS constituent-ils simplement un sous-ensemble de ceux capturés par DYNAMICSLLM ?** Pour répondre à cette question, les événements détectés par chaque outil ont été comparés, en identifiant ceux qui ont été détectés exclusivement par l'un ou l'autre (voir Table 4.4). DYNAMICSLLM a détecté respectivement 318 et 224 événements liés aux défauts de code, en utilisant les configurations 100% LLM et HYBRIDE, qui n'ont pas été détectés par DYNAMICS. À l'inverse, DYNAMICS a identifié respectivement 214 et 193 événements qui n'ont pas été capturés par DYNAMICSLLM

dans les mêmes configurations. De plus, les résultats du test **McNemar** ont mis en évidence une différence statistiquement significative de couverture entre DynamicsLLM (utilisant 100% LLM) et Dynamics (p-value = 0.0000065), indiquant que les différences observées sont peu susceptibles d'être dues au hasard.

Tableau 4.4 Table de contingence des événements liés aux défauts de code entre les configurations de DynamicsLLM et MonkeyRunner, adapté de Cherief *et al.* (2026a)

	Dans MonkeyRunner	Pas dans MonkeyRunner
Dans 100% LLM	556	318
Pas dans 100% LLM	214	3280
Dans Hybride	577	224
Pas dans Hybride	193	3374

Pourquoi Dynamics échoue-t-il à déclencher certains événements liés aux défauts de code, même après avoir exécuté jusqu'à 100× plus d'actions que DynamicsLLM utilisant 100% LLM ? Cela s'explique par sa stratégie d'exploration fondamentalement aléatoire, qui peine à atteindre certaines activités et certains comportements spécifiques. Dans certains cas, l'ordre des actions sur l'interface graphique est crucial. Par exemple, une application jouet a été créée comportant deux activités : la première contient un bouton « Next » sur lequel il faut cliquer 20 fois consécutivement pour accéder à la seconde activité, où se situe le défaut de code. Même après une heure d'exécution, Dynamics n'est pas parvenu à accéder à cette seconde activité.

De plus, la saisie sémantiquement pertinente est essentielle ; il est particulièrement difficile, voire impossible, pour des approches aléatoires de naviguer dans des activités telles que la connexion ou l'inscription, qui nécessitent un remplissage précis des formulaires. Afin d'illustrer davantage cette limitation, une application mobile supplémentaire a été créée, incluant les activités d'enregistrement de compte, de connexion et l'activité principale. Conformément aux résultats précédents, Dynamics n'a pas réussi à atteindre l'activité principale, qui inclut un défaut de code comportemental. En revanche, dans les deux cas, l'outil DynamicsLLM configuré avec 100% LLM a permis de les déclencher avec succès.

Enfin, **quelles sont les limites de l'utilisation des LLM en analyse dynamique ?** D'après les observations, la configuration 100% LLM rencontre parfois des difficultés à atteindre une couverture de test efficace. Par exemple, dans une application de piano comportant de nombreux boutons colorés, aucun des éléments de l'interface utilisateur ne contenait de descriptions textuelles. En l'absence d'un tel contexte textuel, il devient difficile pour le modèle de sélectionner des actions appropriées, car sa prise de décision repose fortement sur des indices sémantiques ; seules 10 actions ont été effectuées en 3 heures de test.

Dans un autre cas, l'interface utilisateur de certaines applications incluait une longue liste verticale d'éléments d'interface graphique, produisant une description excessivement longue dépassant la limite de contexte en entrée du modèle. En conséquence, le contexte tronqué omettait les éléments graphiques situés plus bas dans la liste, empêchant le modèle de raisonner sur ces composants.

Par ailleurs, l'interaction avec les LLM est coûteuse en temps. DynamicsLLM échoue dans certaines tâches complexes. Certains jeux nécessitent l'identification d'un objet dans l'interface utilisateur, et l'outil est alors limité à seulement trois tentatives sur une courte période.

L'utilisation des LLM est également coûteuse en ressources. Cela ne constitue toutefois pas un problème majeur, dans la mesure où les efforts scientifiques et industriels en cours conduisent à des modèles de plus en plus efficaces, avec des exigences matérielles réduites.

> $\boldsymbol{QR_3}$: L'approche basée sur les LLM dépasse les limitations traditionnelles des générateurs de traces d'exécution en produisant des actions sémantiquement pertinentes et logiquement ordonnées. Cependant, son efficacité est limitée par la qualité et l'exhaustivité de la description contextuelle de l'interface utilisateur.

4.4.4 QR_4 : Est-ce que l'approche UCB permet d'améliorer la couverture des événements liés aux défauts de code par rapport à l'approche *hybride heuristique* ?

Motivée par les conclusions de la question de recherche $\boldsymbol{QR_3}$ concernant la contribution du raisonnement basé sur les LLM, et sachant que HYBRIDE repose sur des heuristiques conçues manuellement, cette question est de savoir si une stratégie de prise de décision fondée sur la théorie peut orchestrer de manière adaptative les stratégies d'exploration.

Ainsi, les événements liés aux défauts de code couverts par DYNAMICSLLM sont comparés en utilisant les configurations UCB 02-15 et UCB 02-60 avec la configuration HYBRIDE. Cette comparaison a été réalisée sur 218 applications mobiles présentant au moins un événement lié à un défaut de code. Les autres applications ont été exclues, puisqu'en l'absence d'événements, aucune variation de la couverture ne peut être observée.

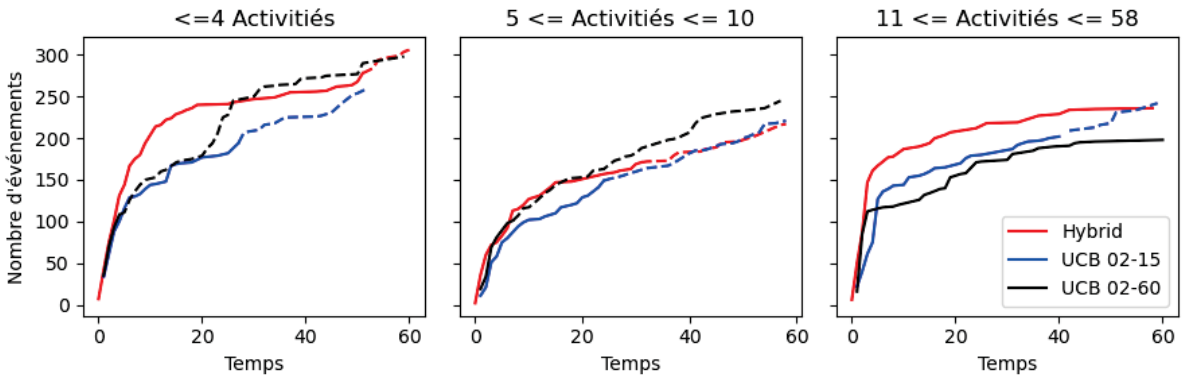

Figure 4.5 Événements liés aux défauts de code couverts par les configurations HYBRIDE, UCB 02-15 et UCB 02-60 au fil du temps. Les lignes pointillées indiquent que certaines exécutions se sont terminées prématurément

La figure 4.5 illustre le nombre d'événements liés aux défauts de code déclenchés au fil du temps par les différentes approches, sur trois catégories d'applications : 124 applications comportant au plus 4 activités, 62 applications comportant entre 5 et 10 activités, et 32 applications comportant plus de 10 activités.

Dans une limite de temps d'une heure, pour les applications comportant peu d'activités, Hybride et UCB 02-60 surpassent UCB 02-15 de 17.2% et 14.2%, respectivement. La différence entre Hybride et UCB 02-60 n'est pas significative selon le test de McNemar (Tableau II-4). Durant les 20 premières minutes, Hybride dépasse les configurations basées sur l'approche UCB qui présentent initialement une couverture similaire. Cette similarité s'explique par la phase d'adaptation des stratégies afin d'identifier la plus performante. Par la suite, UCB 02-60 parvient à réduire rapidement l'écart avec Hybride.

Pour les applications contenant entre 5 et 10 activités, UCB 02-60 atteint une couverture supérieure de 12.9% et 10.9% par rapport à Hybride et UCB 02-15, respectivement. Ces résultats sont statistiquement significatifs (Tableau II-5) et mettent en évidence l'efficacité de UCB 02-60.

Pour les applications comportant plus de 10 activités, UCB 02-60 est moins performant que les autres configurations. Cela peut s'expliquer par la complexité accrue de ces applications, nécessitant un plus grand nombre de prises de décision, comme le permet la configuration UCB 02-15. Cette dernière dépasse légèrement Hybride, mais sans différence statistiquement significative (Tableau II-6).

La figure 4.6 présente le nombre d'événements liés aux défauts de code déclenchés par chaque outil en fonction du nombre d'actions. Les actions générées par les configurations basées sur l'approche *UCB* sont plus efficaces que celles de Hybride. Cette efficacité s'explique par la capacité d'adaptation dynamique de la stratégie sélectionnée au cours de l'exécution.

> $\boldsymbol{QR_4}$: En fonction du temps, UCB dépasse l'approche *heuristique* pour les applications de taille moyenne et présente des performances compétitives pour les petites et les grandes applications. Toutes catégories confondues, l'approche UCB nécessite moins d'actions que l'*heuristique*.

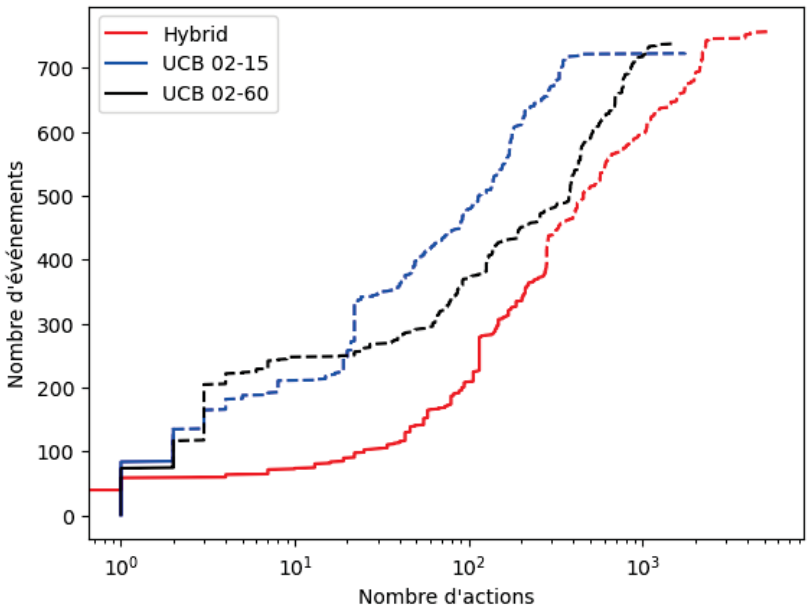

Figure 4.6 Nombre d'événements liés aux défauts de code couverts par les configurations Hybride, UCB 02-15 et UCB 02-60 en fonction du nombre d'actions. Les lignes pointillées indiquent que le temps d'exécution maximal a été atteint pour certaines exécutions

4.5 Menaces à la validité

Bien que cette étude fournisse des informations précieuses sur l'efficacité de DynamicsLLM pour la détection des défauts de code comportementaux, plusieurs menaces à la validité doivent être prises en compte :

Validité interne. Le caractère intrinsèquement aléatoire des LLM et de MonkeyRunner peut influencer les résultats. Étant donné que plusieurs exécutions avec différentes stratégies de génération de traces peuvent produire des résultats variables, l'absence d'expériences répétées en raison de contraintes de temps peut introduire des biais dans les conclusions.

Validité externe. La généralisabilité des résultats à d'autres jeux de données, types d'applications et plateformes constitue une préoccupation majeure. Bien que DynamicsLLM ait été évalué sur 307 applications issues du jeu de données F-DROID, qui couvre une large variété de types

d'applications, celui-ci peut ne pas refléter pleinement la diversité des applications disponibles sur des plateformes telles que le Google Play Store.

Validité de la répétabilité. Un point fort de cette étude réside dans la répétabilité des résultats, puisque tous les outils et jeux de données utilisés sont librement accessibles sur Cherief *et al.* (2025, 2026b). DynamicsLLM et Dynamics, ainsi que les traces d'exécution et les résultats, sont reproductibles, permettant aux futurs chercheurs et praticiens de valider et d'étendre ce travail.

4.6 Conclusion

Ce chapitre a présenté une évaluation expérimentale approfondie de l'approche proposée, visant à mesurer son efficacité pour la génération de traces d'exécution intelligentes et la détection dynamique de défauts de code comportementaux. Quatre questions de recherche ont structuré cette étude et permis d'analyser de manière systématique les performances de DynamicsLLM sous différentes configurations.

Les résultats obtenus pour la première question de recherche démontrent clairement l'efficacité de DynamicsLLM. L'approche atteint une précision de 100% sur l'ensemble des configurations évaluées, tout en maintenant un rappel compétitif. Ces résultats confirment la capacité des traces intelligentes générées à capturer des comportements pertinents pour la détection fiable des défauts de code comportementaux.

La comparaison avec l'outil de l'état de l'art Dynamics, abordée dans la deuxième question de recherche, met en évidence l'apport significatif de l'approche proposée en termes de couverture des événements liés aux défauts de code. En particulier, la configuration 100% LLM permet de couvrir un nombre supérieur d'événements pour un budget d'actions équivalent. L'analyse temporelle montre par ailleurs que la configuration Hybride est mieux adaptée aux applications de petite taille, tandis que la configuration 100% LLM s'avère plus efficace pour des applications de taille intermédiaire. Pour les applications de grande taille, les performances des deux configurations convergent, soulignant la robustesse globale de l'approche proposée.

La troisième question de recherche a permis de mieux caractériser la contribution des LLM à l'efficacité globale de DynamicsLLM. Les résultats montrent que les LLM jouent un rôle clé dans l'exploration guidée des applications, en particulier pour les scénarios complexes, tout en mettant en évidence certaines limites liées à leur coût et à leur dépendance au contexte applicatif.

Enfin, les résultats associés à la quatrième question de recherche confirment l'intérêt de la formalisation de l'approche *hybride* à l'aide de l'algorithme *UCB*. Cette formalisation permet d'améliorer significativement la couverture des événements liés aux défauts de code pour les applications comportant entre cinq et dix activités, tout en maintenant des performances compétitives pour les autres catégories d'applications.

Dans l'ensemble, les résultats expérimentaux valident la pertinence et l'efficacité de l'approche proposée et démontrent son potentiel. Ce chapitre se conclut par une discussion des menaces à la validité, notamment en ce qui concerne la généralisabilité des résultats, l'évaluation ayant été réalisée sur des applications issues de la plateforme F-Droid, ce qui limite leur représentativité vis-à-vis des applications commerciales.

CONCLUSION ET RECOMMANDATIONS

Les applications mobiles deviennent de plus en plus complexes et omniprésentes, et garantir leur qualité constitue un défi majeur. Les défauts de code comportementaux, en particulier, sont difficiles à détecter par l'analyse statique. Bien que DYNAMICS repose sur une approche d'analyse dynamique, il souffre d'un taux élevé de faux négatifs en raison de la faible couverture des générateurs de traces traditionnels, soulignant ainsi la nécessité de méthodes plus intelligentes et adaptatives pour l'analyse dynamique des applications mobiles.

Ce mémoire présente DYNAMICSLLM, un outil qui utilise les LLM pour générer intelligemment des traces d'exécution afin de détecter les défauts de code comportementaux dans les applications mobiles Android. Cet outil améliore l'efficacité des méthodes basées sur l'analyse dynamique en abordant les problèmes des générateurs de traces traditionnels, qui conduisent généralement à une couverture faible et à un taux élevé de faux négatifs.

DYNAMICSLLM a été évalué en utilisant un ensemble de 307 applications mobiles réelles pour comparer ses performances avec l'outil de pointe DYNAMICS. Les résultats montrent que, lorsqu'il est limité à un nombre fixe d'actions, DYNAMICSLLM utilisant l'approche 100% LLM surpasse considérablement DYNAMICS en couvrant les événements liés aux défauts de code. Cependant, lorsque la limite de temps est fixée à une heure, l'approche *hybride heuristique* démontre une couverture rapide pour les applications mobiles contenant peu d'activités, tout en atteignant une couverture compétitive pour les autres catégories.

De plus, les résultats ont mis en évidence le potentiel des approches basées sur LLM, car DYNAMICSLLM a pu déclencher 14% d'événements liés aux défauts de code de plus que l'outil DYNAMICS, soulignant les avantages de la génération de traces d'exécution intelligentes. L'intégration des LLM peut être bénéfique pour d'autres défis de recherche impliquant l'analyse dynamique. Cela inclut les problèmes des applications mobiles tels que la détection des courses de données, le fuzzing et l'analyse des flux de données dans la sécurité mobile et s'étend à

d'autres plateformes. Bien que les LLM restent relativement lents, le domaine évolue rapidement, et des modèles plus rapides et moins gourmands devraient bientôt émerger.

Dans cette optique, la stratégie d'exploration fondée sur UCB a été proposée, afin de dépasser les limites de l'approche *heuristique*, qui repose sur des heuristiques conçues manuellement. Les résultats montrent que l'approche UCB fait 12.9% plus de couverture pour les applications comportant entre 5 et 10 activités, tout en maintenant des performances compétitives pour les autres catégories d'application.

En conclusion, ce travail ouvre une voie prometteuse pour des recherches futures. Les travaux ultérieurs pourraient se concentrer sur le développement de méthodes basées sur les LLM plus rapides, notamment en les combinant avec des approches complémentaires. En particulier, les modèles Vision-Langage (VLM) offrent un potentiel précieux pour l'analyse des éléments d'interface utilisateur non textuels et pour l'amélioration de l'interprétation des retours visuels.

Une autre piste consiste à séparer l'exploration de l'exécution en construisant un modèle partiel ou complet de l'application mobile à tester, puis en le stockant dans un système de type RAG afin d'utiliser une approche target-based pour améliorer l'efficacité. Enfin, une alternative intéressante serait de remplacer le LLM chargé de choisir l'action par un modèle basé sur des représentations vectorielles de texte (text embeddings), prenant en compte l'état actuel, les informations pertinentes et l'objectif. Cette approche permettrait de prioriser les actions candidates selon leur pertinence sémantique, générant ainsi des traces d'exécution sémantiquement significatives, tout en étant beaucoup plus rapide et moins coûteuse que l'utilisation d'un LLM.

ANNEXE I

LES INVITES

```
*** System:
You are a helpful task planner for using an Android mobile application named [Application name].
[...]
(The username and password is what [Persona's name] generally uses for their accounts.)
- name: Jade Green
- phone: 081-827-7650
- [...]
- potential_code_smells: The potential events that can cause the code smell are the following
in format (Type of code smell, class, method):
[
"{'Type': 'HMU_addition','Class': 'a2dp.Vol.service.java', 'Method': 'TextReader'}",[...]
]
[Persona's name]'s ultimate goal is to trigger as many events as possible that can create
potential code  smells.
- [Application name] app has following pages: [ConnectWidgetConfigure, main, ...]
- Currently, [Persona's name]has visited the following pages with the following number of times:
[...]
- Currently, [Persona's name] is on the main page.
- Pages never visited yet: [ConnectWidgetConfigure, ...]
[...].
*** User:
Plan [Persona's name]'s next task based on the following information.
[Persona's name]'s prior knowledge and history of previous tasks so far [...]
[Persona's name]'s learnt knowledge from previous tasks [...]
Current page (organized in a hierarchical structure):
'''json
{
 page_name: main,
 children: [
 {
 ID: 6,
 widget_type: ImageButton,
 content_description: More options,
 possible_action_types: [
 touch,
 long_touch
 ],
 num_prev_actions: 0
 },[...]
 ]
}
'''
Note that 'num_prev_actions' means the number of times the widget has been interacted [...]
[Persona's name] can perform the following types of actions: [...]
I am going to provide a template for your output to reason about your next task step by step
[...]
```

Figure-A I-1 Exemple de prompt envoyé à l'agent *Planner*
tiré de Cherief *et al.* (2025)

```
*** User:
I performed the action, and as a result, the page changed from Main to Media (page changed from
Main to Media)
Events generated are 10
New events generated are 4
All visited events until now are:
[('com.simplemobiletools.gallery.activities.MainActivity.java$_$_findCachedViewById', '0',
'hmuimpl'), ...]
. What should be the next action?
This time, I'll give you the full content of the current page as follows [...]
```json
{
 page_name: Media,
 children: [
 {
 ID: 8,
 widget_type: ImageView,
 content_description: More options,
 possible_action_types: [
 touch,
 long_touch
 ],
 num_prev_actions: 0
 },[...]
 ]
}
```
Guideline for selecting the next action: [...]

Recall that my current task is [...]
This time, I am going to provide a template for your output [...]
```

Figure-A I-2 Exemple de prompt envoyé à l'agent *Observer* tiré de Cherief *et al.* (2025)

ANNEXE II

RÉSULTATS DES TESTS DE MCNEMAR

Tableau-A II-1 Résultats McNemar des configurations Hybride, 100% LLM et MonkeyRunner pour les applications ayant au plus 4 activités
tiré de Cherief *et al.* (2025)

Config. A	Config. B	Commun	Aucun	Juste en A	Juste en B	p-value	Significative
Hybride	100% LLM	216	1070	109	63	0.000452	Oui
Hybride	MonkeyR.	251	1100	74	33	0.000074	Oui
100% LLM	MonkeyR.	201	1096	78	83	0.693540	Non

Tableau-A II-2 Résultats McNemar des configurations Hybride, 100% LLM et MonkeyRunner pour les applications ayant entre 5 et 10 activités
tiré de Cherief *et al.* (2025)

Config. A	Config. B	Commun	Aucun	Juste en A	Juste en B	p-value	Significative
Hybride	100% LLM	151	771	76	115	0.004773	Oui
Hybride	MonkeyR.	134	804	93	82	0.405679	Non
100% LLM	MonkeyR.	149	780	117	67	0.000228	Oui

Tableau-A II-3 Résultats McNemar des configurations Hybride, 100% LLM et MonkeyRunner pour les applications ayant au moins 10 activités
tiré de Cherief *et al.* (2025)

Config. A	Config. B	Commun	Aucun	A seulem.	B seulem.	p-value	Significative
Hybride	100% LLM	196	1461	53	133	0.000000	Oui
Hybride	MonkeyR.	192	1516	57	78	0.070701	Non
100% LLM	MonkeyR.	206	1450	123	64	0.000016	Oui

Tableau-A II-4 Résultats McNemar des configurations Hybride, UCB 02-15 et UCB 02-60 pour les applications ayant au plus 4 activités

Config. A	Config. B	Commun	Aucun	A seulem.	B seulem.	p-value	Significative
Hybride	UCB 02-15	235	795	71	26	0.000005	Oui
Hybride	UCB 02-60	233	756	73	65	0.495868	Non
UCB 02-15	UCB 02-60	202	770	59	96	0.002960	Oui

Tableau-A II-5 Résultats McNemar des configurations Hybride, UCB 02-15 et UCB 02-60 pour les applications ayant entre 5 et 10 activités

Config. A	Config. B	Commun	Aucun	A seulem.	B seulem.	p-value	Significative
Hybride	UCB 02-15	132	1522	85	89	0.761708	Non
Hybride	UCB 02-60	160	1526	57	85	0.018788	Oui
UCB 02-15	UCB 02-60	169	1531	52	76	0.033895	Oui

Tableau-A II-6 Résultats McNemar des configurations Hybride, UCB 02-15 et UCB 02-60 pour les applications ayant au moins 10 activités

Config. A	Config. B	Commun	Aucun	A seulem.	B seulem.	p-value	Significative
Hybride	UCB 02-15	176	956	60	66	0.592980	Non
Hybride	UCB 02-60	157	981	79	41	0.000523	Oui
UCB 02-15	UCB 02-60	157	975	85	41	0.000089	Oui

LISTE DE RÉFÉRENCES

Akinotcho, F., Wei, L. & Rubin, J. (2024). *Mobile Application Coverage : The 30% Curse and Ways Forward*. (Thèse de doctorat, University of British Columbia).

Alzaylaee, M. K., Yerima, S. Y. & Sezer, S. (2017). Improving dynamic analysis of Android apps using hybrid test input generation. *2017 International Conference on Cyber Security and Protection of Digital Services (Cyber Security)*, pp. 1-8. doi : 10.1109/CyberSecPODS.2017.8074845.

Amorim, L., da Costa, I. M., Alves, L. & Figueiredo, E. (2025). Bad Smell Detection using Google Gemini. *2025 IEEE 49th Annual Computers, Software, and Applications Conference (COMPSAC)*, pp. 1637–1642.

Android. (2024). Exécuter des applications sur Android Emulator. Repéré à https://developer.android.com/studio/run/emulator.

Android. (2025a). Android Debug Bridge (adb). Repéré à https://developer.android.com/studio/command-line/adb.

Android. (2025b). Android SDK Command-Line Tools. Repéré à https://developer.android.com/studio/command-line.

Cao, J., Guo, F. & Qu, Y. (2024). JNFuzz-Droid : A Lightweight Fuzzing and Taint Analysis Framework for Android Native Code. *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 255-266. doi : 10.1109/SANER60148.2024.00033.

Cherief, H. A., Avellaneda, F. & Moha, N. (2025). Replication Package. Repéré à https://figshare.com/s/918941f831325d537a57.

Cherief, H. A., Avellaneda, F. & Moha, N. (2026a). DynamicsLLM : a Dynamic Analysis-based Tool for Generating Intelligent Execution Traces Using LLMs to Detect Android Behavioural Code Smells. doi : 10.1145/3793655.3793720.

Cherief, H. A., Avellaneda, F. & Moha, N. (2026b). Replication Package. Repéré à https://figshare.com/s/900725e0ffec6fbaa9d3.

Community, F.-D. (2025). F-Droid : Free and Open Source Android App Repository. Repéré à https://f-droid.org/.

Corporation, O. [Accessed : October 2025]. (2025). Apksigner. Repéré à https://developer.android.com/tools/apksigner.

Developers, A. (2010). The Monkey Tool. Repéré à https://developer.android.com/studio/test/monkey.

Fowler, M. (1999). *Refactoring - Improving the Design of Existing Code* (éd. 1). Addison-Wesley.

Ghafari, M., Gadient, P. & Nierstrasz, O. (2017). Security Smells in Android. *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 121-130.

Guo, X., Qi, X., Li, Y. & Wu, C. (2024). PredRacer : Predictively Detecting Data Races in Android Applications. *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 239-249. doi : 10.1109/SANER60148.2024.00031.

Hallé, S. & Khoury, R. (2017). Event Stream Processing with BeepBeep 3. *RV-CuBES*.

He, X. & openatx. (2024). uiautomator2 Documentation. Repéré à https://github.com/openatx/uiautomator2.

Hecht, G., Rouvoy, R., Moha, N. & Duchien, L. (2015). Detecting antipatterns in Android apps. *2015 2nd ACM international conference on mobile software engineering and systems*, pp. 148–149.

Khan, M. U., Lee, S. U.-J., Abbas, S., Abbas, A. & Bashir, A. K. (2021a). Detecting Wake Lock Leaks in Android Apps Using Machine Learning. *IEEE Access*, 9, 125753-125767. doi : 10.1109/ACCESS.2021.3110244.

Khan, M. U., Lee, S. U.-J., Wu, Z. & Abbas, S. (2021b). Wake lock leak detection in Android apps using multi-layer perceptron. *Electronics*, 10(18), 2211.

Li, Y., Yang, Z., Guo, Y. & Chen, X. (2017). Droidbot : a lightweight UI-guided test input generator for Android. *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pp. 23–26.

Li, Y., Yang, Z., Guo, Y. & Chen, X. (2019). Humanoid : A Deep Learning-Based Approach to Automated Black-box Android App Testing. *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1070-1073. doi : 10.1109/ASE.2019.00104.

Liu, T., Wang, H., Li, L., Bai, G., Guo, Y. & Xu, G. (2019). DaPanda : Detecting Aggressive Push Notifications in Android Apps. *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 66-78. doi : 10.1109/ASE.2019.00017.

Liu, Z., Chen, C., Wang, J., Chen, M., Wu, B., Che, X., Wang, D. & Wang, Q. (2023). Chatting with GPT-3 for Zero-Shot Human-Like Mobile Automated GUI Testing. Repéré à https://arxiv.org/abs/2305.09434.

Manukulasooriya, G., Munasingha, R., Fernando, H., Uthpali, H., Yeshani, A. & Siriwardana, D. (2024). Enhancing Automated Android Application Security through Hybrid Static and Dynamic Analysis Techniques. *2024 International Congress on Human-Computer Interaction, Optimization and Robotic Applications (HORA)*, pp. 1-6. doi : 10.1109/HORA61326.2024.10550843.

Mesbah, D., El Madhoun, N., Al Agha, K. & Chalouati, H. (2025). Leveraging prompt-based large language models for code smell detection : a comparative study on the MLCQ dataset. *International Conference on Emerging Internet, Data & Web Technologies*, pp. 444–454.

Novendra, R. D. & Danar Sunindyo, W. (2024). Emerging Trends in Code Quality : Introducing Kotlin-Specific Bad Smell Detection Tool for Android Apps. *IEEE Access*, 12, 63895-63903. doi : 10.1109/ACCESS.2024.3397055.

Ollama. (2024). Get up and running with LLMs. Repéré à https://ollama.com.

OpenAI. (2025). API reference. Repéré à https://platform.openai.com/docs/api-reference/responses/create#responses-create-temperature.

Palomba, F., Di Nucci, D., Panichella, A., Zaidman, A. & De Lucia, A. (2017). Lightweight detection of Android-specific code smells : The ADoctor project. *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 487–491.

Prestat, D., Moha, N. & Villemaire, R. (2022). An empirical study of Android behavioural code smells detection. *Empirical Software Engineering*, 27(7), 179. doi : 10.1007/s10664-022-10212-8.

Prestat, D., Moha, N., Villemaire, R. & Avellaneda, F. (2024). DynAMICS : A Tool-Based Method for the Specification and Dynamic Detection of Android Behavioral Code Smells. *IEEE Transactions on Software Engineering*, 50(4), 765-784. doi : 10.1109/TSE.2024.3363223.

Reimann, J., Brylski, M. & Aßmann, U. (2014). A tool-supported quality smell catalogue for Android developers. *Softwaretechnik-Trends Band 34, Heft 2*.

Sadik, A. R. & Govind, S. (2025). Benchmarking LLM for Code Smells Detection : OpenAI GPT-4.0 vs DeepSeek-V3. Repéré à https://arxiv.org/abs/2504.16027.

statista.com. (2022a). App Stores and Marketplaces. Repéré à https://www.statista.com/topics/1729/app-stores/.

statista.com. (2022b). Google Play App Downloads and Revenue Statistics. Repéré à https://www.statista.com/statistics/734332/google-play-app-installs-per-year/.

Tsutano, Y., Bachala, S., Srisa-an, W., Rothermel, G. & Dinh, J. (2019). Jitana : A modern hybrid program analysis framework for Android platforms. *Journal of Computer Languages*, 52, 55-71. doi : https://doi.org/10.1016/j.cola.2018.12.004.

Vallée-Rai, R., Co, P., Gagnon, E. M., Hendren, L., Lam, P. & Sundaresan, V. (2010). Soot : a Java bytecode optimization framework. *Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2005)*, pp. 249–264. doi : 10.1145/1094811.1094838.

Wang, J., Huang, Y., Chen, C., Liu, Z., Wang, S. & Wang, Q. (2024). Software Testing With Large Language Models : Survey, Landscape, and Vision. *IEEE Transactions on Software Engineering*, 50(4), 911-936. doi : 10.1109/TSE.2024.3368208.

Wen, H., Wang, H., Liu, J. & Li, Y. (2024). DroidBot-GPT : GPT-powered UI Automation for Android. Repéré à https://arxiv.org/abs/2304.07061.

Wu, D., Mu, F., Shi, L., Guo, Z., Liu, K., Zhuang, W., Zhong, Y. & Zhang, L. (2024). iSMELL : Assembling LLMs with Expert Toolsets for Code Smell Detection and Refactoring. *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, (ASE '24), 1345–1357. doi : 10.1145/3691620.3695508.

Wu, Z., Chen, X. & Lee, S. U.-J. (2023). A systematic literature review on Android-specific smells. *Journal of Systems and Software*, 201, 111677. doi : https://doi.org/10.1016/j.jss.2023.111677.

Yasin, H. N., Hamid, S. H. A. & Raja Yusof, R. J. (2021). Droidbotx : Test case generation tool for Android applications using Q-learning. *Symmetry*, 13(2), 310.

Yoon, J., Feldt, R. & Yoo, S. (2024). Intent-Driven Mobile GUI Testing with Autonomous Large Language Model Agents. *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pp. 129–139.

Yunmar, R. A., Kusumawardani, S. S., Widyawan & Mohsen, F. (2024). Hybrid Android Malware Detection : A Review of Heuristic-Based Approach. *IEEE Access*, 12, 41255-41286. doi : 10.1109/ACCESS.2024.3377658.