UNIVERSITÉ DU QUÉBEC

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

COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN MATHÉMATIQUES ET INFORMATIQUE
APPLIQUÉES

PAR
THIERNO ALIOU BA

PRÉDICTION DE DÉFAUTS LOGICIELS GRÂCE AUX GRANDS MODÉLES DE LANGAGE

AVRIL 2025

REMERCIEMENTS

Avant tout, je tiens à exprimer ma profonde gratitude envers Dieu, qui m'a accordé la force, la patience et la persévérance nécessaires pour mener à bien ce projet. Sans Sa grâce, rien n'aurait été possible. Je Lui suis reconnaissant pour la lumière qui m'a guidée tout au long de ce parcours, et j'ai foi qu'Elle continuera de m'éclairer dans mes projets futurs.

Je souhaite exprimer ma reconnaissance la plus sincère à Dr Fadel Touré et Dr Usef Faghihi, mes directeurs de recherche. Leur encadrement rigoureux, leur disponibilité et leurs conseils avisés ont été déterminants dans la réalisation de ce mémoire. Travailler sous leur supervision a été un privilège, et leur confiance m'a constamment motivée à donner le meilleur de moi-même.

Mes pensées les plus affectueuses vont à mes parents, mes frères et ma sœur, pour leur amour inconditionnel, leur soutien moral et leurs encouragements constants. Leur présence, même à distance, a été une source précieuse de réconfort et d'équilibre tout au long de ce parcours exigeant.

Je tiens également à remercier mes collègues et partenaires de projet, avec qui j'ai eu le plaisir de collaborer. Leurs échanges stimulants, leur esprit d'équipe et leur expertise ont enrichi cette expérience de recherche et m'ont permis de grandir tant sur le plan académique que personnel.

Enfin, je remercie toutes les personnes qui, de près ou de loin, ont contribué à la réussite de ce projet. Cette aventure m'a apporté de précieuses leçons et des compétences durables, et je suis impatiente de poursuivre ma contribution à la recherche dans mon domaine.

RÉSUME

La qualité logicielle est essentielle au bon fonctionnement des systèmes critiques, où les défauts peuvent entraîner des conséquences graves. Or, la détection précoce des fautes reste un défi majeur, en raison de la complexité croissante des logiciels et du coût élevé des corrections tardives.

Ce mémoire propose une approche de prédiction de défauts basée sur les LLMs, en analysant directement le code source sans recourir à des artefacts intermédiaires tels que les métriques logicielles. Les modèles BigBird et StarCoder ont été testés, mais leurs performances se sont révélées limitées.

Le modèle DeepSeek-Coder, pré-entraîné sur du code source et optimisé pour la classification, a été retenu. Il permet l'analyse complète de classes Java longues et complexes, avec l'ajout de mots (tokens) contextuels pour améliorer la généralisation entre projets.

Les expérimentations menées sur dix projets Java open source de la base de données PROMISE ont montré des résultats solides : F1-score moyen de 0.80 et AUC moyenne de 0.89.

Ces résultats confirment l'intérêt des LLMs spécialisés pour améliorer la fiabilité logicielle et réduire les coûts de maintenance.

Mots-clés : qualité logicielle, prédiction de défauts, deepSeek-coder, llm, code, apprentissage profond, validation inter-projets

ABSTRACT

Software quality is essential for the reliable operation of critical systems, where bugs can lead to serious consequences. However, early fault detection remains a major challenge due to increasing software complexity and the high cost of late-stage corrections.

This thesis proposes a bug prediction approach based on Large Language Models (LLM), by analyzing raw source code directly—without relying on intermediate artifacts such as software metrics. The BigBird and StarCoder models were evaluated but showed limited performance.

The DeepSeek-Coder model, pre-trained on source code and optimized for classification tasks, was selected. It enables the analysis of full-length, complex Java classes, with the integration of project-specific context tokens to improve cross-project generalization.

Experiments conducted on ten open-source Java projects from the PROMISE dataset demonstrated strong results: an average F1-score of 0.80 and an average AUC of 0.89.

These findings highlight the potential of specialized LLMs to enhance software reliability and reduce maintenance costs.

Keywords: software quality, bug prediction, deepSeek-coder, llm, code analysis, deep learning, cross-project validation

LISTE DES ABREBIATIONS

AUC	Area Under the Curve
AST	Abstract Syntax Tree
BERT	Bidirectional Encoder Representations from Transformers
CFG	Control Flow Graph
CK	Chidamber and Kemerer
CNN	Convolutional Neural Network
DFG	Data Flow Graph
DNN	Dense Neural Network
DL	Deep Learning
FIM	Fill-in-the-Middle
FPR	False Positive Rate
FN	False Negative
FP	False Positive
GGNN	Gated Graph Neural Network
GQA	Grouped-Query Attention
ISO	International Organization for Standardization
LLM	Large Language Model
LOC	Lines of Code
LSTM	Long Short-Term Memory
MAE	Mean Absolute Error
MLP	Multilayer Perceptron
PDG	Program Dependency Graph
RFC	Response for a Class
RoPE	Rotary Positional Embedding
SMOTE	Synthetic Minority Oversampling Technique
SVM	Support Vector Machine
TN	True Negative
TP	True Positive
TPR	True Positive Rate

WMC Weighted Methods per Class

LISTE DES FIGURES

LISTE DES TABLEAUX

TABLE DE MATIERES

CHAPITRE 1
INTRODUCTION

1.1. Contexte et Motivation

La qualité logicielle constitue un pilier essentiel dans le développement de systèmes robustes et fiables, en particulier dans des domaines critiques tels que la santé, les infrastructures financières ou les transports. Dans ces secteurs, une défaillance logicielle peut entraîner des conséquences graves, allant de la perte de données sensibles à des interruptions de service majeures, voire à des risques pour la sécurité humaine. Il est donc impératif de maintenir un taux de défauts aussi faible que possible tout au long du cycle de développement.

L'un des défis majeurs réside dans la détection précoce des défauts logiciels. Plus un défaut est identifié tard dans le processus de développement, plus son coût de correction est élevé. En effet, les activités de test peuvent représenter jusqu'à 75 % du coût total du développement logiciel [1].

Dans ce contexte, les Grands Modèles de Langage (Large Language Models, LLMs) et les techniques de l'Apprentissage Profond (Deep Learning, DL) ouvrent de nouvelles perspectives. Les LLMs sont désormais capables de générer du code, de proposer des corrections et même de générer des cas de test. Ces capacités permettent de soutenir certaines étapes du développement logiciel, notamment en facilitant la prédiction automatique de défauts, ce qui peut réduire les coûts tout en améliorant la fiabilité des logiciels.

L'application du DL à la qualité logicielle vise ainsi à concevoir des outils capables d'identifier les défauts dès les premières phases du cycle de vie logiciel, renforçant la fiabilité des applications et répondant aux exigences de sécurité et de performance des environnements critiques.

1.2. Problématique

La prédiction des défauts logiciels représente un enjeu majeur dans le développement logiciel moderne. Cette problématique peut être analysée selon deux axes complémentaires :

- Analyse du code source : Extraire des informations syntaxiques et structurelles pertinentes à partir du code, sans dépendre exclusivement de métriques manuelles ou de graphes intermédiaires.
- Détection précoce des défauts : Identifier les composants défectueux le plus tôt possible afin de réduire drastiquement les coûts de maintenance et prévenir les défaillances en production [2].

Bien que plusieurs approches aient été proposées dans la littérature pour aborder ces aspects, des limitations subsistent encore, notamment en matière de généralisation inter-projets et de précision des prédictions.

1.3. Objectif de recherche

Ce mémoire vise principalement à améliorer la prédiction des défauts dans le code source Java à l'aide de LLMs.

L'objectif principal consiste à développer une méthode efficace de classification binaire (avec défaut / sans défaut) des classes Java, permettant d'identifier les défauts logiciels de manière précoce et fiable. Pour cela, nous exploitons des LLMs fondés sur l'architecture à auto-attention [3], capables d'extraire des représentations syntaxiques et contextuelles riches directement à partir du code source brut, sans recourir à des représentations intermédiaires telles que des métriques ou des graphes.

Un objectif secondaire consiste à concevoir et entraîner un système prédictif capable d'optimiser l'allocation des ressources dédiées aux activités de test et de maintenance, en concentrant les efforts sur les composants les plus susceptibles de contenir des défauts.

Ces objectifs visent à renforcer la qualité du code tout en diminuant les coûts de développement, en s'appuyant sur des techniques d'apprentissage profond pour soutenir le développement logiciel moderne.

1.4. Structure du mémoire

Ce mémoire est structuré en plusieurs chapitres, chacun abordant un aspect clé de notre recherche, suivis d'une conclusion.

Le premier chapitre présente le contexte général de la recherche, la problématique abordée, ainsi que les objectifs poursuivis.

Le deuxième chapitre est consacré à la revue de littérature. Il regroupe les travaux relatifs aux métriques logicielles, aux approches traditionnelles de prédiction de défauts, ainsi qu'à l'émergence des modèles de langage appliqués au code.

Le troisième chapitre est constitué de l'article principal intitulé *"Software Defects Prediction : Code Is All You Need"*, dans lequel sont décrits la méthodologie adoptée, les expérimentations réalisées, ainsi que les résultats obtenus à l'aide du modèle DeepSeek-Coder.

Le quatrième chapitre propose une discussion générale des résultats, en mettant en évidence les limites des approches antérieures, les apports du modèle retenu et les perspectives.

Enfin, la conclusion résume les contributions principales de cette recherche et présente les pistes d'amélioration et de recherche à explorer.

CHAPITRE 2
ÉTAT DE L'ART

2.1. Dimensions de la qualité logicielle et importance de la prédiction des défauts

La qualité logicielle regroupe un ensemble de caractéristiques essentielles permettant d'évaluer dans quelle mesure un logiciel satisfait aux exigences fonctionnelles et non fonctionnelles. Selon le modèle ISO/IEC 25010 [4, 5], elle s'articule autour de plusieurs dimensions clés, dont certaines sont directement impactées par la présence ou l'anticipation des défauts.

Parmi ces dimensions, la maintenabilité désigne la capacité à modifier facilement un logiciel pour corriger des erreurs ou répondre à de nouvelles exigences. Elle est influencée par des facteurs tels que la complexité, la modularité et la cohésion du code [6].

La fiabilité mesure la capacité du logiciel à fonctionner sans défaillance dans des conditions spécifiques. Les défauts non détectés compromettent directement cette stabilité [7].

La sécurité concerne la protection contre les intrusions et les défaillances liées aux erreurs de programmation. De nombreux rapports montrent que la majorité des vulnérabilités logicielles sont dues à des défauts [8].

La performance désigne l'aptitude du logiciel à répondre rapidement tout en optimisant les ressources utilisées. Des erreurs dans la gestion mémoire ou les algorithmes peuvent nuire aux performances [9].

Enfin, l'utilisabilité reflète la facilité d'usage d'un système. Des comportements inattendus causés par des défauts peuvent altérer l'expérience utilisateur [10].

La prédiction des défauts contribue principalement à l'amélioration de la fiabilité logicielle, en permettant d'anticiper les défaillances et de cibler les efforts de test. Elle peut également soutenir, de manière indirecte, d'autres dimensions telles que la maintenabilité, en facilitant la détection de modules problématiques, ce qui permet de réduire les coûts de maintenance. En effet, un défaut corrigé en production peut coûter jusqu'à 100 fois plus cher qu'en phase de conception [1].

Les métriques logicielles sont couramment utilisées pour quantifier ces dimensions et identifier les composants à risque. Leur intégration dans les modèles de prédiction permet une évaluation plus objective et proactive de la qualité logicielle.

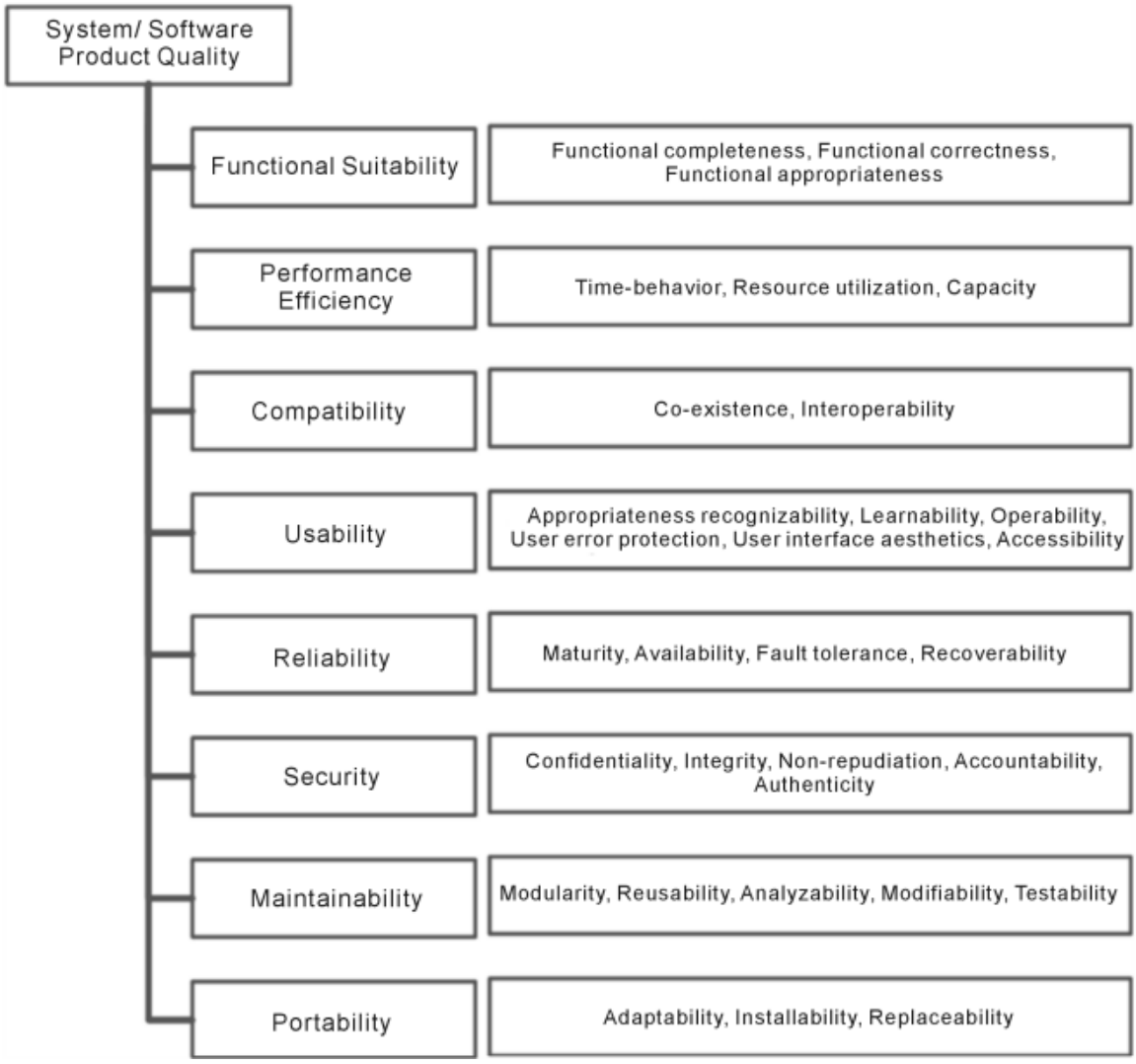

Figure 2.1: Le modèle de qualité logicielle ISO/IEC 25010 tiré de [5]

2.2. Métriques logicielles

L'analyse des métriques logicielles constitue une approche fondamentale pour évaluer la qualité interne d'un système logiciel et détecter les composants susceptibles de contenir des défauts. Ces métriques permettent de quantifier objectivement des aspects structurels

du code, facilitant ainsi la prédiction des défauts par des modèles d'apprentissage automatique ou profond.

Parmi les métriques les plus utilisées dans la littérature, on retrouve notamment :

2.2.1. La suite de métriques Chidamber-Kemerer (CK)

Introduite en 1994 , la suite CK [11] est spécifiquement conçue pour les systèmes orientés objet. Ces indicateurs mesurent la complexité, le couplage, l'héritage et la cohésion, qui sont des facteurs clés associés à la maintenabilité et à la probabilité d'erreurs.

2.2.2. La suite de métriques Bansiya

Proposée comme une extension des métriques CK, la suite Bansiya [12] introduit des mesures supplémentaires axées sur la qualité orientée objet, telles que la densité fonctionnelle, le facteur de réutilisabilité ou encore l'efficacité. Elle permet d'élargir la couverture des dimensions de qualité logicielle tout en conservant une compatibilité avec les outils d'analyse de code existants.

2.2.3. Les métriques de complexité de système

Ces métriques globales [13] visent à évaluer la complexité d'un système dans son ensemble, en tenant compte de la taille, de la profondeur des appels, du nombre de dépendances entre modules ou encore de la connectivité du graphe d'appel. Une complexité excessive est souvent corrélée à une plus grande vulnérabilité aux défauts.

2.2.4. Les métriques Halstead

Basée sur une approche empirique, la suite de Halstead [14] repose sur le comptage des opérateurs et des opérandes dans le code source. Elle permet d'estimer l'effort cognitif nécessaire à la compréhension d'un programme, la densité d'information, et la difficulté de maintenance, trois aspects étroitement liés à la présence de défauts.

Les métriques logicielles capturent des caractéristiques structurelles du code telles que la complexité, le couplage ou la cohésion, qui sont souvent liées à l'introduction de défauts. De nombreuses études empiriques ont démontré que des valeurs extrêmes dans ces métriques sont des indicateurs fiables de défauts. Leur utilisation dans les modèles de prédiction permet d'identifier les composants à risque, de cibler les efforts de test et de réduire les coûts de maintenance, tout en renforçant la fiabilité globale du logiciel.

2.3. Les représentations graphiques du code

Les représentations graphiques du code modélisent les relations syntaxiques, sémantiques ou structurelles entre différentes entités d'un programme. Chaque représentation repose sur un graphe, où les nœuds symbolisent des éléments du code (instructions, blocs, variables) et les arêtes traduisent des relations (hiérarchie, flux d'exécution, dépendance, ordre).

2.3.1. L'arbre de syntaxe abstraite (AST)

L'AST [15] représente la structure grammaticale du code. Les nœuds correspondent aux unités syntaxiques (déclarations, expressions), et les arêtes expriment leur imbrication hiérarchique.

2.3.2. Le graphe de flot de contrôle (CFG)

Le CFG [16] décrit les chemins possibles d'exécution d'un programme. Les nœuds sont des blocs de code, et les arêtes indiquent les transitions entre blocs selon la logique de contrôle (conditions, boucles).

2.3.3. Le graphe de dépendances de données (DFG)

Le DFG [17] illustre comment les données circulent entre instructions. Les nœuds représentent des opérations ou des variables, et les arêtes montrent les relations de production et d'utilisation.

2.3.4. Le graphe de dépendance du programme (PDG)

Le PDG [17, 18] capture à la fois les dépendances de données et de contrôle dans un programme. Les nœuds représentent des instructions (assignations, conditions, appels), et les arêtes traduisent les relations de dépendance : les arêtes de données relient les instructions partageant des variables, tandis que les arêtes de contrôle relient une instruction conditionnelle à celles qu'elle gouverne. Cette structure permet de modéliser avec précision les interactions logiques complexes et s'avère utile pour identifier des erreurs liées à la propagation des valeurs ou à l'ordre d'exécution.

2.4. Travaux connexes

La prédiction des défauts logiciels vise à identifier automatiquement les composants du code source susceptibles de contenir des défauts. Deux grandes familles d'approches se

distinguent dans la littérature : celles fondées sur les métriques logicielles et celles exploitant des représentations graphiques du code. Les performances de ces travaux sont souvent évaluées à l'aide de métriques standards telles que le F1-score et l'AUC (ces métriques sont détaillées dans l'article scientifique, section 3.4.3.).

2.4.1. Approches basées sur les métriques logicielles

Les métriques logicielles sont souvent utilisées comme variables explicatives dans les modèles de classification.

Dans leurs travaux, Okutan et al. [19] ont utilisé un modèle bayésien pour identifier les métriques les plus influentes sur les défauts logiciels. Lines Of Code (LOC), Response For Class (RFC) et Lines Of Commented Code (LOCQ) se sont révélées prédictives, tandis que Depth of Inheritance Tree (DIT) et Number of Children (NOC) étaient moins informatives.

Le travail de Subhan et al. [20] ont conçu un modèle Convolutional Neural Networks–Long Short-Term Memory (CNN-LSTM) utilisant uniquement des métriques extraites de grands jeux de données de vulnérabilités. Leur modèle atteint un F1-score de 0.95, illustrant la puissance des métriques combinées au DL.

Esteves et al. [21] ont proposé US-XGBoost, un modèle léger fondé sur la sélection aléatoire de sous-ensembles de métriques. Il obtient une AUC de 0.89 tout en offrant une interprétabilité accrue grâce aux valeurs SHAP.

Siddiqui et al. [22] ont amélioré les performances de modèles de DL standards en appliquant la méthode d'équilibrage Synthetic Minority Oversampling Technique (SMOTE). Sur la base de données NASA CM1, ils obtiennent des F1-scores proches de 0.97.

Aydin et al. [23] ont introduit des métriques spécifiques aux types d'erreurs (syntaxiques vs. Logiques), afin de mieux différencier les défaillances. Leur modèle Dense Neural Network (DNN) a significativement réduit les faux positifs.

Les approches basées sur les métriques logicielles réduisent le code à des statistiques globales, ignorant la sémantique et la structure interne des programmes. Leur pouvoir prédictif est limité, parfois redondant ou non généralisable entre projets [19].

2.4.2. Approches basées sur les représentations graphiques du code

Ces approches exploitent la structure syntaxique ou logique du code source, permettant une compréhension plus fine de son comportement.

Seml [24], un modèle LSTM basé sur des séquences de tokens[1], extraits des arbres d'AST. Il atteint un F1-score moyen de 0.77, démontrant l'intérêt des représentations sémantiques.

Zhou et al. [25] ont proposé Devign, un modèle de détection de vulnérabilités. Devign combine l'AST, le CFG, le DFG et la séquence naturelle du code. Ces différentes perspectives sont intégrées dans un réseau de graphes récurrents, enrichi d'une couche de convolution spécialisée pour effectuer la classification au niveau fonctionnel. Le modèle a atteint un F1-score moyen de 73,26 %.

Mou et al. [26] ont introduit Tree-Based Convolutional Neural Network (TBCNN), un modèle de réseau de neurones convolutifs spécialement conçu pour exploiter la structure des AST dans les langages de programmation. Chaque nœud de l'arbre est représenté sous forme vectorielle, sur lequel le modèle applique des convolutions adaptées à la structure arborescente, suivies d'un pooling dynamique. Le modèle atteint un F1-score de 89,7 %.

Allamanis et al. [27] ont proposé un modèle fondé sur les Gated Graph Neural Networks (GGNN) pour apprendre des représentations sémantiques de programmes à partir de graphes enrichis. Ces graphes combinent à la fois des relations syntaxiques et sémantiques offrant ainsi une vision fine de la structure et du comportement du code source. Le modèle a atteint une précision de 78,2 % sur la détection d'une mauvaise utilisation d'une variable dans le code source.

Bien que plus expressives, ces représentations entraînent une perte d'information liée au prétraitement, sont coûteuses à générer, et restent incomplètes [25-27].

Face à ces limites, les LLMs pré-entraînés soit sur des langages naturels ou sur du code source offrent une alternative efficace : ils capturent la syntaxe, et se généralisent mieux aux projets variés [28, 29].

[1] Un *token* est une unité de texte (mot, sous-mot ou caractère) utilisée comme entrée pour les modèles de langage afin d'analyser ou générer du texte.

2.5. LLM

2.5.1. Mécanisme d'auto-attention

Ce mécanisme permet au modèle de capturer les dépendances entre les différents éléments d'une séquence, quelle que soit leur distance relative.

2.5.1.1. Fonctionnement de l'auto-attention

L'auto-attention consiste à pondérer dynamiquement chaque mot d'une séquence en fonction de tous les autres mots, afin de produire une représentation contextuelle enrichie. Ce processus repose sur trois vecteurs pour chaque mot : Query (Q)–Requête ;

Key (K)–Clé; Value (V)–Valeur associée. La sortie est une moyenne pondérée des vecteurs V, où les poids sont calculés par une fonction de compatibilité entre Q et K. L'attention est formellement définie par :

$$Attention(Q, K, V) = softmax\left(\frac{(Q\ K^{\mathrm{T}})}{\sqrt{d_k}}\right)V$$

Où $\sqrt{d_k}$ est la dimension des vecteurs de clés, utilisée pour stabiliser les gradients. La fonction *softmax*[2] transforme les scores d'attention en probabilités normalisées, permettant au modèle de pondérer l'importance relative de chaque mot dans une séquence lors du calcul des représentations contextuelles.

2.5.1.2. Multi-head attention

Le Transformer emploie une stratégie dite de multi-head attention, qui consiste à appliquer un nombre *h* de mécanismes d'attention en parallèle, chacun capturant des relations différentes dans des sous-espaces de représentation. Les résultats des têtes sont concaténés et projetés pour produire la sortie finale.

Cette approche permet au modèle de focaliser son attention sur plusieurs types de dépendances et apprendre des représentations plus riches et expressives.

[2] La fonction *softmax* transforme un vecteur de scores en probabilités normalisées dont la somme est égale à 1, permettant ainsi d'interpréter la sortie d'un modèle comme une distribution sur plusieurs classes.

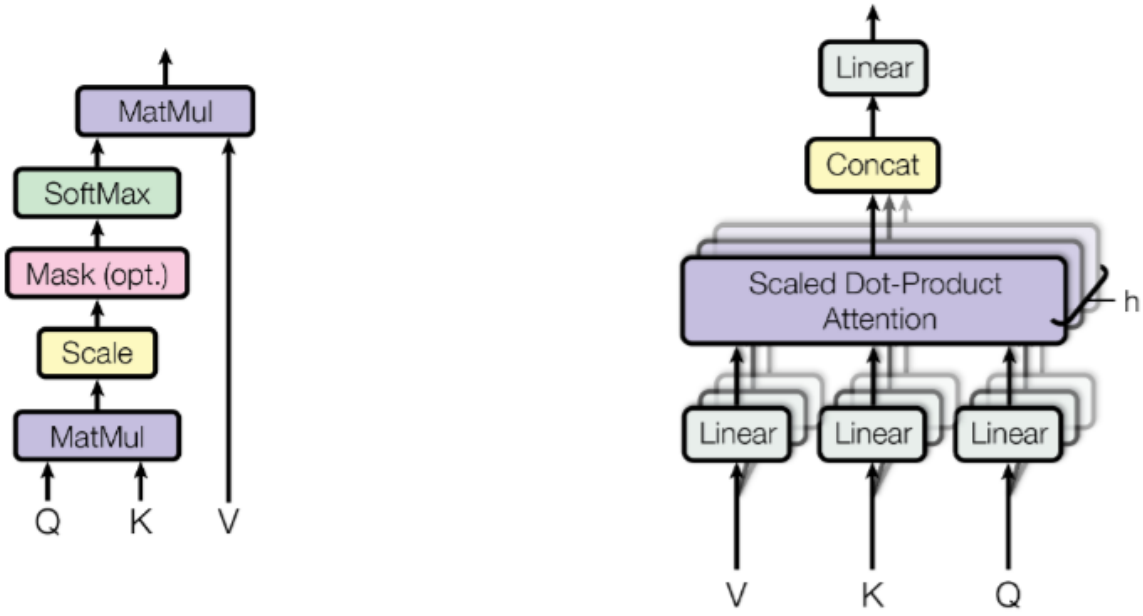

Figure 2.2 Mécanismes d'Auto-attention (à gauche) et de Multi-head Attention (à droite) tirés de [3]

L'auto-attention présente plusieurs avantages :

- Parallélisation complète des calculs pendant l'entraînement ;
- Complexité constante en nombre d'opérations pour relier deux éléments distants de la séquence ;
- Chemins d'information courts, facilitant l'apprentissage de dépendances à longue portée.

Ces propriétés en font une architecture de choix pour les tâches de traitement du langage naturel et l'analyse de code source.

2.5.2. BigBird

Les modèles Transformers classiques, comme Bert [30], présentent une complexité quadratique en mémoire et en calcul, ce qui limite leur capacité à traiter des séquences longues (souvent inferieures 1024 tokens). Ce plafond est problématique pour les tâches nécessitant un contexte étendu, comme l'analyse de classes entières. BigBird [31] répond à cette limite en introduisant un mécanisme d'attention clairsemée (sparse attention), permettant de réduire la complexité de $O^{(2n)}$ à $O(n)$ et maintenant le pouvoir expressif du Transformer. BigBird combine trois types d'attention complémentaires :

(1) Fenêtre locale (local window) : Chaque token est relié à ses voisins immédiats dans une fenêtre de taille w. Cela permet de capturer les dépendances locales importantes, comme les relations entre instructions consécutives.

(2) Attention aléatoire (random attention) : Chaque token est relié à un petit nombre de tokens choisis aléatoirement dans la séquence. Cela assure une connectivité globale, maintenant une faible distance moyenne entre les tokens dans le graphe d'attention.

(3) Attention globale (global attention) : Certains tokens ont une attention sur toute la séquence et sont visibles par tous les autres tokens. Ces tokens agissent comme des relais pour agréger et redistribuer l'information à grande échelle.

Ce schéma permet à BigBird de simuler un Transformer dense tout en étant plus rapide et moins gourmand en mémoire, et il conserve théoriquement la complétude des chemins attentionnels.

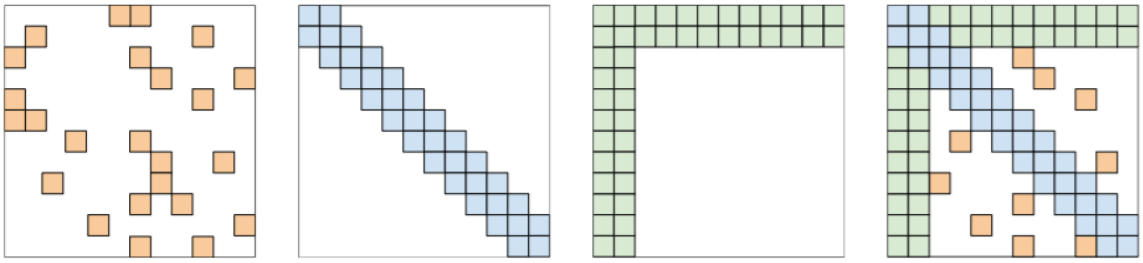

Figure 2.3: Éléments constitutifs du mécanisme d'attention utilisé dans le modèle BigBird tirés de [31]

Dans chaque élément de la Figure 2.3, la couleur blanche indique l'absence d'attention. (a) Attention aléatoire, (b) Attention par fenêtre glissante, (c) Attention globale. (d) Modèle combiné BigBird.

2.5.2.1. Avantages pour l'analyse de classes

La taille moyenne d'une classe peut dépasser largement les 512 tokens, surtout lorsqu'elle contient plusieurs méthodes. Le mécanisme de BigBird permet : une représentation complète du code sans troncature ; la capture simultanée des dépendances locales (entre

lignes de code voisines) et globales (entre méthodes ou blocs distants) ; une efficacité computationnelle accrue, rendant l'entraînement plus viable sur GPU classiques.

Ces propriétés en font un candidat naturel pour des tâches telles que la prédiction de métriques logicielles ou de défauts dans le code source complet d'une classe.

2.5.2.2. Limites pour la prédiction de défauts logiciels

Malgré ses avantages, BigBird présente certaines limites dans le contexte du code :

- Pré-entraînement sur du texte naturel : BigBird a été pré-entraîné sur des corpus textuels génériques, et non sur du code source, ce qui limite sa capacité à reconnaître des motifs syntaxiques propres à la programmation.
- Absence de biais inductifs spécifiques au code : Contrairement à des modèles comme CodeBERT [32] pré-entrainé sur des segments de code, BigBird ne tient pas compte de la structure syntaxique propre au code, comme les blocs, les appels de fonctions ou les dépendances de données.

2.5.3. StarCoder

StarCoder [33] est un modèle de langage conçu pour comprendre, générer et transformer du code source, en s'appuyant sur des données massives issues de dépôts GitHub publics. Contrairement aux modèles généralistes, StarCoder est pré-entraîné exclusivement sur du code, ce qui le rend particulièrement adapté aux tâches de compréhension du code.

Plusieurs éléments font de StarCoder un modèle performant pour traiter des fichiers de code longs, tels que des classes :

- Pré-entraînement sur un corpus multilingue contenant plus de 80 langages de programmation, totalisant plus d'un trillion de tokens extraits de dépôts GitHub publics.
- Fine-tuning intensif sur Python : une phase d'entraînement supplémentaire sur 35 milliards de tokens Python, améliorant les performances sur des tâches spécifiques telles que l'auto-complétion, l'infilling ou la génération de code robuste.

- Fenêtre contextuelle étendue : avec une capacité d'analyse allant jusqu'à 8192 tokens, rendue possible par l'intégration de FlashAttention [34] et Multi-Query Attention (MQA) [35], StarCoder est capable d'encoder des classes entières.
- Infilling (FIM) : le modèle est entraîné à compléter du code non seulement en fin de séquence, mais aussi au milieu, grâce à un schéma d'apprentissage orienté Fill-in-the-Middle, qui renforce sa capacité à modéliser les dépendances syntaxiques et contextuelles [33].

2.5.3.1. Avantages pour l'analyse de classes

Grâce à sa large fenêtre contextuelle et à sa préformation sur du code structuré, StarCoder peut :

- Encoder l'ensemble du contenu d'une classe ;
- Capturer les dépendances inter-méthodes ou inter-blocs de la classe ;
- Identifier des structures inhabituelles ou suspectes qui pourraient indiquer la présence de défauts.

2.5.3.2. Limites pour la prédiction de défauts logiciels

Malgré ses avantages, StarCoder présente certaines limites pour la tâche spécifique de prédiction de défauts logiciels :

- Objectif d'origine différent : StarCoder est conçu pour la génération de code et la complétion, mais pas spécifiquement pour la classification binaire ou la prédiction de défauts. Il ne bénéficie donc pas d'un entraînement supervisé sur des données annotées avec des défauts.
- Biais de langage : Une part importante du prétraitement est concentrée sur Python, ce qui peut désavantager d'autres langages lors de tâches complexes comme la prédiction de défauts.

2.5.4. DeepSeek-Coder

DeepSeek-Coder [36] est un modèle de langage open-source allant de 1,3 à 33 milliards de paramètres, conçus spécifiquement pour le traitement du code source. Contrairement aux modèles généralistes, DeepSeek-Coder est pré-entraîné intégralement à partir de zéro sur

un corpus de 2000 milliards de tokens issus de dépôts GitHub organisés à l'échelle du projet, couvrant 87 langages de programmation. Cette approche ambitieuse vise à doter le modèle d'une compréhension fine de la structure, des dépendances et des motifs syntaxiques du code.

DeepSeek-Coder intègre plusieurs innovations architecturales et méthodologiques :

- Pré-entraînement à l'échelle du dépôt (repo-level pretraining) : Contrairement aux modèles pré-entraînés sur des fichiers isolés, DeepSeek-Coder aligne les fichiers d'un même projet selon leurs dépendances, ce qui renforce sa capacité à comprendre les interactions entre classes et méthodes dans un projet complet [36].
- Fenêtre contextuelle étendue à 16 000 tokens : Grâce à une adaptation des Rotary Positional Embeddings (RoPE) [37] et à l'utilisation de FlashAttention v2 [34], le modèle peut ingérer le code complet d'une grande classe, ce qui est fondamental pour l'analyse de structure et la détection de motifs anormaux.
- Apprentissage par inférence en FIM : Le modèle est entraîné à reconstituer une portion manquante dans le code, entre un préfixe et un suffixe donnés. Cette capacité améliore la compréhension bidirectionnelle et la robustesse du modèle dans des scénarios complexes [38].
- Optimisation de l'attention : DeepSeek-Coder utilise des mécanismes d'attention avancés tels que Grouped-Query Attention (GQA) [39] pour améliorer l'efficacité de calcul tout en maintenant la précision sur de longues séquences.

2.5.4.1. Avantages pour l'analyse de classes

- D'analyser intégralement une classe ;
- De capter des relations complexes entre les blocs de code distants, utiles pour identifier les défauts potentiels liés à la logique ou à l'architecture ;
- De bénéficier d'un pré-entraînement spécialisé sur des projets réels, augmentant la sensibilité aux patterns de bugs récurrents.

Malgré ses avantages, DeepSeek-Coder n'est pas directement optimisé pour la classification binaire (défectueux vs. Non défectueux). Un fine-tuning spécifique est nécessaire pour la prédiction de défauts.

Parmi les trois modèles comparés, DeepSeek-Coder se distingue par sa capacité à capturer des motifs propices aux défauts (defect-leading patterns), grâce à trois facteurs clés :

- Pré-entraînement à l'échelle du dépôt : Contrairement à BigBird (entraîné sur du texte naturel) et StarCoder (entraîné sur des fichiers de code isolés), DeepSeek-Coder est pré-entraîné sur des projets complets. Cela lui permet de modéliser les interactions entre classes, méthodes et fichiers, essentielles pour détecter des motifs liés aux défauts logiciels.
- Fenêtre contextuelle étendue (jusqu'à 16 000 tokens) : Supérieure à celle de StarCoder (jusqu'à 8k) et plus adaptée que celle de BigBird, cette capacité permet à DeepSeek-Coder d'analyser une classe entière sans perte de contexte, tout en capturant les dépendances à longue portée.
- Objectif d'apprentissage orienté code : Grâce au FIM et à son focus sur la compréhension structurelle du code, DeepSeek-Coder est plus sensible aux irrégularités logiques et aux structures anormales caractéristiques des défauts.

En résumé, sa spécialisation sur le code, son apprentissage contextuel profond et sa capacité à traiter de longues séquences font de DeepSeek-Coder un modèle plus prometteur que BigBird ou StarCoder pour analyser le code et prédire les défauts logiciels.

La section suivante présente notre article scientifique centré sur son expérimentation approfondie, détaillant la méthodologie, les résultats et les apports de ce modèle pour la prédiction de défauts.

CHAPITRE 3
L'ARTICLE SCIENTIFIQUE

Le présent chapitre constitue le cœur du mémoire sous forme d'article scientifique, tel que soumis à la conférence QRS. Il présente les résultats principaux de cette recherche, centrée sur l'utilisation du LLM DeepSeek-Coder pour la prédiction de défauts logiciels à partir du code source brut.

L'article ci-dessous, intitulé "*Software Defects Prediction: Source Code Is All You Need*", explore les capacités du modèle à détecter les classes Java potentiellement défectueuses, sans recourir à des représentations intermédiaires. Il est co-écrit par Thierno Aliou BA (80 %), Fadel Toure (10 %) et Usef Faghihi (10 %).

Cet article est actuellement soumis à la 25e conférence internationale de Qualité, fiabilité et sécurité logicielles (QRS) en Chine.

SOFTWARE DEFECTS PREDICTION:
SOURCE CODE IS ALL YOU NEED

Thierno Aliou BA, Fadel Toure, and Usef Faghihi
Department of Computer Science, University of Quebec at Trois-Rivières, Quebec, Canada
thierno.aliou.ba@uqtr.ca, fadel.toure@uqtr.ca, usef.faghihi@uqtr.ca

Abstract— Software defects prediction is a complex task. Its effective achievement could improve system reliability and prevent costly consequences for the industry by focusing quality assurance efforts on critical components. Traditional approaches typically use models trained on source code metrics or other artifacts as proxies to simplify training but can result in a loss of valuable information and limit generalizations. In this study, we propose a novel defects prediction model, based on DeepSeek-Coder configured with Llama-based sequence classification, analyzing raw source code without intermediate artifacts. Our approach leverages DeepSeek-Coder's ability to process long sequences of tokens, enabling a comprehensive analysis of object-oriented components and capturing

complex syntactic and semantic structures. The model we built was trained and evaluated on 10 Java open-source projects using the "Leave One Out" project cross-validation technique. The results we obtained are very encouraging in terms of precision (0.93), recall (0.97), F1 score (0.95) and AUC (0.98) and suggest that raw source code feeding fine-tuned LLMs is a viable and robust alternative to traditional defect prediction methods.

Keywords- Software defects prediction; Machine learning; Large code models; Ssource code analysis; Object-oriented systems

3.1. INTRODUCTION

The ability to predict component defects helps focus insurance quality efforts (as unit tests), on critical components to detect, locate and fix defects. It reduces maintenance costs but also enhances the overall reliability and security of software before deployment. Undetected software defects can lead to security vulnerabilities, critical runtime failures, and significant financial losses for organizations [1].

Software defects have resulted in catastrophic consequences in the recent past. In 2020, a software error in a medical assistance system led to incorrect insulin dosages, exposing diabetic patients to severe hypoglycemia risks [2]. In 2022, a defect in the United States kidney transplant registry resulted in incorrect patient rankings, delaying or even preventing life-saving organ transplants [3]. Similarly, in 2012, a malfunction in an automated trading algorithm caused $440 million in losses within 45 minutes, leading to the bankruptcy of Knight Capital [4]. Additionally, in 2003, a defect in an electric grid management system failed to detect an overload, triggering a massive blackout that affected 55 million people across the United States and Canada [5]. These incidents highlight the critical impact of software defects on financial and operational stability and on human lives. The economic burden of late software defects detection is substantial, with costs varying depending on the severity of the defect and the industry. The expenses related to debugging, testing, and verification are estimated to represent between 50% and 75% of the total budget for software development projects, exceeding $100 billion each year [6]. The total cost of debugging remains very high due to the increasing complexity of systems.

Existing approaches to predict defects often rely on software metrics and graph-based representations of code. While these methods have been effective, they may suffer from several key limitations:

(1) Limited Generalization – Many models depend on domain-specific datasets, making them less adaptable to new projects and diverse codebases [7][8].

(2) High Dependency on Feature Engineering – Metric-based methods may require extensive manual feature extraction, which varies across projects and development environments [7][9].

(3) Computational Complexity – Some techniques use graph-based representations (e.g., Abstract Syntax Trees (AST), dependency graphs), which introduce significant computational overhead and hinder scalability [10].

(4) Interpretability Challenges – Many advanced models lack explainability, making them difficult to adopt in real-world software development pipelines [11][12].

Metrics, graphs and other artifacts extracted from source code play proxy roles. Those artifacts can be seen as static embedding spaces manually built to simplify the datasets since they aren't learned and optimized during model training phases. Furthermore, relying on them could drastically reduce the valuable information contained in the original source code.

Given these limitations, we present a new approach that leverages recent advances in Machine Learning (ML) and Deep Learning (DL) to analyze raw source code without intermediate transformations or artifact extraction.

Rather than claiming to automate multiple steps of software development, we propose a fine-tuned Large Language Model (LLM) based on DeepSeek-Coder [13] that we specifically designed for software defects prediction. Our approach focuses on one critical aspect: defect prediction from raw source code. Unlike traditional models DeepSeek-Coder directly analyzes raw source code. The approach uses advanced Natural Language Processing (NLP) techniques and DL architectures, particularly the Transformers model [14][15]. The DeepSeek-Coder model is pre-trained on large-scale datasets of source code. Its ability to process long token sequences enables a more complete and fine-grained

analysis of component source code, capturing patterns associated with software defects [13].

The primary objective of this study is to propose a reliable model of software defects prediction fed by raw source code, bypassing the artifact extraction step and which takes advantage of full information from the source code without loss. Specifically, this research aims to:

- Optimize DeepSeek-Coder for software defects prediction using raw source code analysis.
- Evaluate the model's generalization capability using the leave one out system cross validation on multiple open-source projects.
- Analyze the benefits and limitations of raw source code analysis without artificial embedding (manually built intermediate code representations).

The study makes the following major contributions:

(1) Building a Direct Raw Source Code Analysis: Unlike traditional models, our approach eliminates dependency on manually engineered artifacts as proxies, processing raw source code.

(2) Leveraging DeepSeek-Coder Backbone with Long Sequence Capability: Enabling the model to handle sequences up to 16,000 tokens, to be able to analyze most of the entire Java class files, capturing syntactic and semantic patterns without truncation [13].

(3) Introducing Project Contextual Token to embed global project-level context, improving model generalization across varied projects with differing coding styles and conventions.

(4) Validating model using the leave one out cross-project validation on PROMISE[3] database and achieving 0.93 of precision, 0.97 of recall, 0.95 of F1-score, and 0.98 of AUC indicating excellent generalizability.

[3] https://github.com/feiwww/PROMISE-backup

(5) Handling Class Imbalance by adopting a selective undersampling strategy to address the imbalance between defective and non-defective classes.

By demonstrating the feasibility of LLM-based defect prediction, this study paves the way for a more efficient, scalable, and adaptable defect prediction approach in modern software development environments.

3.2. RELATED WORK

3.2.1. Approaches of Software Defects Prediction

Software defects prediction is a key research area aimed at identifying portions of the source code likely to contain defects. Various approaches have been proposed, which can be broadly categorized into three main groups: (1) Metric-based models and (2) Graph-based code representations, and (3) Direct source code-based models.

3.2.2. Metric-Based Models

Software metrics have long been used in defect prediction as they provide a quantifiable measure of code source attributes related to high-level software quality. Commonly used metrics in prediction approach for object-oriented systems include traditional metrics like source lines of code (SLOC), Chidamber-Kemerer [16] metric suite, Bansiya metric suite [17], system complexity metrics [18, 19], and Halstead metrics [19]. These features are often used to train machine learning algorithms such as Random Forest, Support Vector Machines (SVMs), Decision Trees, and Neural Networks for defect prediction [20] [21].

Aydin, Phung and Ogunshile [22] improved this approach by incorporating error-specific measures (e.g. distinguishing between syntax errors and logical errors). His DNN-based model achieved an f1-score between 0.70 – 0.88 and significantly reduced false positives compared to classical methods.

Siddiqui and Mustaqee [23] addressed software quality improvement through defect prediction using machine learning combined with data balancing. To mitigate overfitting caused by class imbalances, they applied SMOTE to the NASA CM1 dataset and evaluated multiple classifiers. Among them, Random Forest emerges as a standout performer, with an accuracy of 0.98 and an F1-score of 0.97. SVM and KNN also demonstrate high accuracy rates of 0.98 and 0.97, respectively, while LDA shows balanced performance with

an accuracy of 0.96 and an F1-score of 0.96. This study demonstrates the effectiveness of balancing techniques in enhancing predictive performance for software defects prediction.

Esteves, Figueiredo, Veloso, Viggiato, and Zivian [24] proposed US-XGBoost, a defect prediction approach that leverages randomized feature subsets with XGBoost to enhance both performance and interpretability. Applied to eight Jureczko datasets, the method reached an AUC of 0.89 and an F1-score of 0.69 using as few as 1 to 3 features. By integrating SHAP values, the study identified key project-specific metrics (e.g. SLOC, NPM, AMC), highlighting that lightweight models can remain both accurate and interpretable.

Subhan, Wu, Bo, Sun and Rahman [25] proposed a hybrid CNN–LSTM model for software vulnerability detection using 18 code source metrics as features. Evaluated on a large dataset derived from SySeVR and SARD, the model achieved an F1-score of 0.95. The CNN–LSTM architecture also achieved the lowest false positive (2.54%) and false negative (8.1%) rates. Results confirm the effectiveness of code metrics for deep learning-based vulnerability detection, especially when combined with sequential modeling.

Despite their effectiveness, metric-based models face key limitations. Abstracting code into numerical metrics results in loss of structural and semantic information, limiting predictive power. These models rely on proxy features that require expert-driven selection and often fail to generalize across projects. Their inability to capture code meaning hinders detection of complex, context-dependent defects.

3.2.3. Source Code Representations

To overcome the limitations of metric-based approaches, researchers have explored representations of code, such as: Abstract Syntax Trees (ASTs), Control Flow Graphs (CFGs), ASCII.

Liang, Yu, Jiang and Xie [26] proposed Semantic LSTM for Defect Prediction (Seml), a deep learning model that combines AST-based token sequences with pretrained word embeddings and a sequential LSTM network. Trained on 13 open-source Java projects, Seml achieved F1-scores ranging from 0.71 to 0.85 across projects, in within-project defect prediction (WPDP). In cross-project defect prediction (CPDP), it reported F1-scores

between 0.46 and 0.75. These results highlight the effectiveness of semantic-aware token embeddings combined with LSTM-based modeling.

Pan, Lu, Xu and Gao [27] proposed an improved CNN model using AST-based token sequences and skip-gram word embeddings to enhance within-project defect prediction (WPDP). Evaluated on the expanded PROMISE Source Code (PSC) dataset, the model achieved an average F1-score of 0.57. The study introduced the concept of hyperparameter instability, highlighting its dual role as a challenge and an opportunity in deep learning-based defect prediction.

Chen et al. [28] proposed Deep Transfer Learning for Defect Prediction (DTL-DP), an end-to-end deep learning framework that visualizes ASCII values of source code as RGB images and applies transfer learning with a self-attention mechanism. Using a modified AlexNet architecture and Multiple Kernel Maximum Mean Discrepancy (MK-MMD) to align feature distributions, the model avoids traditional feature engineering by working directly on raw code. Evaluated on 10 PROMISE Java projects, DTL-DP achieving average F1-scores of 0.64 in Within-Project Defect Prediction (WPDP) and 0.61 in Cross-Project Defect Prediction (CPDP). Their findings highlight the benefits of combining software visualization with domain-adaptive deep learning.

Phan, Nguyen and Bui [29] proposed a graph-based defect prediction model that converts source code into control flow graphs (CFGs) extracted from assembly code and applies a multi-view, multi-layer directed graph-based convolutional neural networks (DGCNN). Evaluated on four real-world CodeChef datasets, the model outperformed AST-based and metric-based baselines, achieving up to 0.84 accuracy and 0.82 AUC. The approach demonstrates the advantages of leveraging semantic execution flows over syntactic structures for defect prediction.

Pan, Lu, and Xu [30] conducted an empirical study on software defects prediction using CodeBERT, a pre-trained language model for programming languages. They introduced four variants: CodeBERT-NT (trained from scratch), CodeBERT-PT (fine-tuned), CodeBERT-PS (sentence-based), and CodeBERT-PK (keyword-based). Experiments on cross-version (CVPSC) and cross-project (CPPSC) PROMISE datasets showed that pre-trained CodeBERT-PT outperformed CodeBERT-NT by up to 2.1% F1-score in cross-

version and 0.9% in cross-project tasks. Additionally, CodeBERT-PS and CodeBERT-PK outperformed the traditional CodeBERT-PT, achieving average F1-scores of 0.61 (CVPSC) and 0.57 (CPPSC), highlighting the impact of prediction pattern design in enhancing performance.[30]

Uddin et al. [31] proposed SDP-BB, a defect prediction model combining a pre-trained BERT encoder with a BiLSTM and attention mechanism to capture semantic and contextual code features. Evaluated on 10 PROMISE datasets in WPDP and CPDP settings, SDP-BB achieving up to 0.77 F1-score (WPDP) and 0.58 F1-score (CPDP). The study also demonstrated that data augmentation and full-token input strategies improve model performance over AST-node-based approach. [31]

The approaches based on source code representation also face notable limitations. Transforming code into ASTs, CFGs, embeddings, or images often introduces structural bias and overlooks deep semantics and logic. These representations can be computationally expensive and may break contextual continuity, weakening the model's understanding of the source code. The approaches also struggle to generalize across diverse source codes and often lack interpretability due to their black-box nature.

3.2.4. Limitations of Existing Models

Despite significant advancements in software defects prediction, existing approaches suffer from the following key limitations.

3.2.4.1. Class Imbalance in Datasets

A fundamental challenge in defect prediction is the class imbalance problem. In most software projects, defective files constitute a small fraction of the overall codebase. Machine learning models are sensitive to that bias, leading to poor recall rates for defect prediction [1]. Several techniques have been explored to mitigate the bias including (1) undersampling: which aims at reducing the number of majority class instances [32], (2) oversampling: which duplicates instances of the minority class to balance the dataset [33], and (3) Cost-sensitive learning that assigning higher weights to classification errors [34]. However, these mitigating strategies have their drawbacks. Undersampling may remove valuable data, while oversampling increases the risk of overfitting. Cost-sensitive learning

also requires careful tuning of hyperparameters, which may vary significantly between projects [25].

3.2.4.2. Information Loss and Code Abstraction

Many traditional approaches rely on hand-engineered software metrics (e.g., CK metrics, SLOC, RFC) or intermediate representations such as abstract syntax trees (ASTs), control flow graphs (CFGs), or token sequences. While these abstractions help simplify the modeling process, they introduce a significant information loss. Metrics only capture high-level statistical properties of the code and often ignore deeper semantic structures, such as data dependencies, control logic, or inter-method interactions [35]. Similarly, representations like ASTs and CFGs are limited to syntactic or control-level views of the code, overlooking rich semantic cues present in the raw source [26]. Moreover, preprocessing techniques such as identifier removal or code-to-image transformations (e.g., ASCII encoding) can discard naming conventions, comments, and structural nuances critical for understanding software behavior [28]. These simplifications hinder the model's ability to fully learn latent defect patterns, particularly in complex or large-scale systems [27].

3.2.4.3. Limited Generalization Capability

Many models perform well within a specific project but fail to generalize across different software projects. Among the reasons we find: (1) the variability in coding styles implemented by different teams using different conventions and idioms; (2) The differences in software architectures, the source code has unique structural properties affecting defect distribution; And finally, (3) the dataset specificity (Project sizes, open-source aspects of projects…) making models unreliable for large-scale industry applications. Investigations have been conducted to improve cross-projects generalization. In 2024, Bal and Kumar's proposed a cross-project CNN model, which has shown promising results, but its effectiveness remains limited [25].

3.2.4.4. Computational Overhead and Integration Complexity

Metric-based and graph-based models require significant preprocessing, including static extraction of source code metrics and dependency graph construction, which is

computationally expensive, especially for large-scale software systems. These complexities hinder their integration into CI/CD pipelines, where real-time feedback is essential [28][36].

Large language models (LLMs) have recently shown remarkable success across a wide range of software engineering tasks, including code summarization, vulnerability detection, bug fixing, and code generation. Unlike traditional approaches based on handcrafted metrics or intermediate representations (e.g., ASTs, CFGs), LLMs operate directly on raw source code, enabling them to capture both syntactic patterns while preserving the full context of the code. Models such as CodeBERT [37], PLBART [38], and CodeX [39] have achieved state-of-the-art results in tasks like code completion and vulnerability identification.

Building on this trend, our study proposes a fine-tuned DeepSeek-Coder, a Transformer model fed by raw source code, capable of defect prediction. It simplifies deployment and scalability and enhances the generalization across projects, addressing the long-standing limitations of existing methods.

Table 3.1: Existing Bug Prediction Approaches

Articles	**Data**	**Model Type**	**F1-score**	**Disadvantages**
[22][23] [24][25]	Source code software metrics	Random Forest, SVM, KNN, DNN, CNN, LSTM	0.69 – 0.97	Dependent on extracted metrics, Information Loss, poor cross-project generalization
[26] [27] [28][29] [30][31]	Source code derived Graph	CNN, LSTM, GNN, LLMs	0.57 – 0.85	Information Loss, High computational cost, requires advanced extraction steps

3.3. METHODOLOGY

3.3.1. Data Overview

The current study uses ten open-source projects from PROMISE [40] a publicly available repository and widely used in software defects prediction studies [41][42][43][44][45][46].

This dataset includes the source code of different OO projects, each project related to 2 CSV files. The first file lists the fully qualified software class names with several metrics and labelled by its number of defects. The second file contains the ASTs of the Java classes. The data is from multiple open-source Java projects spanning various domains providing an ideal framework for evaluating the generalization ability of our approach. Provided metrics and AST are not utilized in our study. We focused on the Class package which allows retrieving the matching source code of the class; and the number of defects that we converted into a binary label indicating the presence (1) or absence (0) of defects.

The class distribution reveals a clear predominance of non-defective classes. Table 3.2 shows for each of ten projects, the number of Defective Classes (DC), Non-Defective Classes (NDC) and the Ratio (NDC/DC) which values range from 0.73 to 4.77. This global imbalance ratio of 1.8 highlights the need for an appropriate preprocessing strategy to avoid bias during the trained phases.

Table 3.2: Projects defects distribution

Projects	**DC**	**NDC**	**Ratio:**
Ant	350	1342	3.83
Camel	562	2222	3.95
Jedit	303	1446	4.77
Log4j	260	189	0.73
Lucene	438	344	0.79
Poi	707	671	0.95
Synapse	162	473	2.92
Velocity	367	272	0.74
Xalan	1806	1514	0.84
Xerces	217	838	3.86
Total	5172	9311	1.8

3.3.2. Data Preprocessing

To ensure high data quality and improve the model's generalization capabilities, several preprocessing operations were implemented.

3.3.2.1. Removal of Irrelevant Textual Information

Since the model is designed to analyze the structure and intrinsic logic of the source code exclusively, we removed or replaced potentially biased textual information: (1) All comments were removed, following the Krinke's recommendations [46], to prevent the

model from learning explicit human annotations. (2) All string literals were replaced with a generic token (STRING) to ensure that the model does not learn to identify project-specific characteristics [47]. These transformations force the model to learn exclusively from the intrinsic characteristics of the source code.

3.3.2.2. Source Code Standardization

To ensure a uniform representation of the source code across all projects and eliminate irrelevant stylistic variations, we applied automatic formatting using Google Java Formatter[4]. This tool standardizes: (1) Indentation, spaces, and Java language style conventions. (2) Removal of minor syntactic differences and stylistic noise, allowing the model to focus only on structural and semantic patterns relevant to defect detection.

This standardization step promotes effective model generalization and significantly reduces bias stemming from individual developer or team styles.

3.3.2.3. Feature Selection

To ensure that our approach relies solely on the raw source code and binary label of defects, we used the class fully qualified name in the first CSV file and *javalang parser*[5] tool to insert the matching class source code in the final dataset.

3.3.2.4. Class Deduplication in Training Datasets

To prevent bias caused by code repetitions, particularly present in multiple versions of the same project, a rigorous duplication removing step was carried out. Following the recommendations of the study [48], we identified and removed duplicated classes across different versions of the same project. We eliminated strict duplicates to retain only one representative occurrence of each class. Table 3.3 shows the defective class distribution after the deduplication step.

[4] https://github.com/google/google-java-format
[5] https://github.com/javaparser/javaparser

Table 3.3: Projects defects distribution after deduplication

Project	DC	NDC	Ratio
Ant	323	963	2.98
Camel	448	1200	2.68
Jedit	263	902	3.43
Log4j	175	106	0.61
Lucene	355	179	0.50
Poi	599	227	0.38
Synapse	140	341	2.43
Velocity	248	171	0.69
Xalan	524	739	1.41
Xerces	204	66	0.32
Total	3279	4894	1.49

3.3.2.5. Inclusion of Specific Tokens

To explicitly integrate the context of each project without modifying the internal structure of the code, a specific token identifying each project was inserted at the beginning of each source file (PROJECT_1, PROJECT_2, etc.).

This technique is inspired from studies [37] and [49], which showed that adding general contextual information significantly improves the generalization capability of source code-based models.

3.3.2.6. Handling Class Imbalance in Training Datasets

The imbalance between defective and non-defective classes presents a major challenge, which can lead to significant bias in predictions. This phenomenon, well documented by He and Garcia [50], requires an appropriate rebalancing strategy.

We adopted a selective under sampling approach, directly inspired by the recommendations of studies [51] and [52]. This strategy includes (1) Gradual reduction of the majority class instances (non-defective) to achieve a balanced ratio (50/50); and (2) Prioritizing reduction within the most imbalanced projects to preserve as much intrinsic data diversity as possible.

3.3.3. Model Architecture

3.3.3.1. Overview of DeepSeek-Coder

The architecture of DeepSeek-Coder relies on two key mechanisms to effectively capture the characteristics of source code:

- Rotary Position Embeddings (RoPE): This technique improves the handling of long sequences by encoding the relative position of tokens. RoPE allows the model to better capture long-term dependencies, which are essential for identifying complex interactions across the codebase.[13] [53]
- Grouped-Query Attention: This mechanism increases the speed of the attention process by grouping queries, allowing the model to efficiently process complex and large structures while accelerating inference [13] [54].

These two mechanisms help DeepSeek-Coder to process sequences up to 16,000 tokens long, which is particularly relevant for in-depth analysis of complete Java classes.

3.3.3.2. Model Fine-Tuning

To fine-tune the DeepSeek-Coder model, we used LLaMA [55] based architecture for sequence classification (LlamaForSequenceClassification[6]) [59]. The model uses the last token [CLS] to perform the classification, as used in other causal models such as GPT-3.5 [56]. We choose the 1.3-billion parameter variant, DeepSeek-Coder 1.3B, for fine-tuning.

3.3.3.3. Model Configuration

To maximize the fine-tuning efficiency, the base configuration has been adjusted as following: The Sequence length is set to 2048 tokens, providing sufficient coverage to capture important dependencies between code segments. The dropout of 0.2 prevents overfitting by introducing light but effective regularization [57]. 2 output labels to match binary defective classification. Finally, a learning rate of $3e^{-5}$ for fine-tuning Transformers of similar size [58]. This configuration ensures an ideal balance between model performance and computational constraints.

[6] https://huggingface.co/docs/transformers/model_doc/llama#transformers LlamaForSequenceClassification

3.3.3.4. Training Parameters

Training was conducted following an iterative hyperparameter optimization approach, including:

- Optimizer: AdamW, particularly suited for robust weight management during Transformer fine-tuning [59].
- Scheduler: Cosine Annealing, which gradually adjusts the learning rate to ensure stable and fast convergence [60].
- Weight decay: 0.01, to prevent overfitting by regularizing the model's weights [57].
- Mixed precision (FP16): Used to optimize GPU memory utilization and accelerate computations while preserving model accuracy [61].
- Gradient accumulation: To optimize GPU resource utilization, a gradient accumulation mechanism over 16 steps was applied, effectively increasing the batch size without exceeding available GPU memory [62].
- Number of epochs: 20 epochs.

This setup allows the model to learn from complex data patterns while maintaining computational efficiency.

3.3.4. Code Tokenization

Tokenization is a critical step in the fine-tuning process. This phase involves using DeepSeek-Coder's proprietary tokenizer with the addition of context special tokens (*[PROJECT_i], $i \in [1;10]$*) to effectively capture the global context of each project. These specific tokens allow the model to better generalize on unseen projects by learning differentiated contexts. To ensure uniform input size during training, all sequences exceeding the 2048 tokens were truncated on the right side, while shorter sequences were padded using the [*PAD*][7] token. An analysis of token distribution (Table 3.4) showed that most Java classes have a token length between 500 and 1800 tokens, with less than 5% exceeding the 2048-token limit. Setting the sequence length to 2048 tokens allows the

[7] https://huggingface.co/docs/tokenizers/index

model to capture most of the code structures without losing significant information, while maintaining manageable computational costs.

This token limit ensures that the model can process most Java classes while avoiding unnecessary padding or loss of information due to truncation.

Table 3.4: Distribution of number of Tokens per classes

# of Tokens	Percentage of Classes (%)
[0, 500)	22%
[500, 1000)	35%
[1000, 1500)	25%
[1500, 2048)	13%
+2048	5%

3.4. EXPERIMENTS

This section describes the experimental protocol used to evaluate the ability of the DeepSeek-Coder model to predict the presence of defects in OO source code. We tested two main configurations: (1) raw source code and (2) raw source code, including project contextual tokens.

3.4.1. Experimental Configuration

We used the Leave One Project Out Cross Validation (LOPOCV) to test the model's generalization ability on unseen projects. This approach simulates a realistic scenario where the model is exposed to unknown projects in production, when trained on other available system histories. It thereby increases the practical relevance of the evaluation.

The LOPOCV consists of using one project as the test dataset while combined remaining projects are used for training (80%) and validation (20%). The training dataset is rebalanced using a selective undersampling strategy. The reduction is performed primarily in projects with the highest imbalance.

- Extreme imbalance: Some projects are heavily dominated by a single class, such as Poi (majority defective) or Camel (majority non-defective).
- Inverted projects: Projects such as Log4j or Velocity show a reversal of the majority class between training and test phases.

- Balanced projects: No project is perfectly balanced, but some projects such as Xalan or Synapse show more homogeneous distribution.

The distribution of classes in the dataset (Table 3.2) showed significant imbalance between DC and NDC, with a predominance of non-defective classes (NDC) in most projects.
The test datasets are kept imbalanced and the number of NDC is generally higher than the number of DC.

3.4.2. Tested Scenarios

We conducted two main experiments to analyze the model's ability to learn defect-leading patterns directly from the source code and to be aware of project-specific context.

Scenario 1—Prediction Based Solely on Raw Source Code: This scenario evaluates the model's ability to predict the presence of defects based only on the syntactic and semantic structure of the code.

Scenario 2—Prediction Based on Source Code with Project Context: This scenario tests whether adding project token-context allows the model to better capture project-specific patterns and improve the model's generalization capability.

3.4.3. Evaluation Metrics

The model's performance was evaluated using several standard classification metrics.

- False Positive (FP): Number of non-defective classes incorrectly predicted as defective.
- False Negative (FN): Number of defective classes incorrectly predicted as non-defective.
- True Negative (TN): Number of non-defective classes correctly predicted.
- True Positive (TP): Number of defective classes correctly predicted.

In binary classification, the decision threshold determines the conversion of the model logit into binary predictions. In this study, we use a default threshold of 0.5.

Table 3.5 : Confusion matrix

	Predicted Defective	**Predicted Non-Defective**
Defective	TP	FN
Non-Defective	FP	TN

3.4.3.1. Precision

Precision measures the model's ability to avoid false positives, i.e., the number of correct positive predictions relative to the total number of positive predictions made.

$$Precision = \frac{TP}{\mathrm{TP} + \mathrm{FP}}$$

3.4.3.2. Recall

Recall evaluates the model's ability to correctly detect actual bugs, i.e., the number of correctly predicted positive cases relative to the total number of actual positive cases.

$$Recall = \frac{TP}{\mathrm{TP} + \mathrm{FN}}$$

3.4.3.3. F1-Score

The F1-score represents the harmonic meaning between precision and recall. It helps balance the model's ability to avoid false positives and detect actual bugs.

$$F_1 = \frac{2 \times \mathrm{Précision} * \mathrm{Recall}}{\mathrm{Precision} + \mathrm{Recall}}$$

3.4.3.4. AUC (Area Under ROC Curve)

The ROC (Receiver Operating Characteristic) curve measures the model's ability to distinguish between defective and non-defective classes. It plots the true positive rate (TPR) against the false positive rate (FPR) at different thresholds values.

The AUC corresponds to the area under the ROC curve, which represents the true positive rate as a function of the false positive rate.

$$AUC = \int_0^1 TPR\,(FPR) d(FPR)$$

where:

$$\text{TPR (True Positive Rate)} = \frac{TP}{\text{TP} + \text{FN}}$$

$$\text{FPR (False Positive Rate)} = \frac{FP}{\text{FP} + \text{TN}}$$

Threshold optimization can be performed using the ROC or the PR curves, selecting the threshold that maximizes both sensitivity and specificity, or the F1-score to achieve a better balance score between precision and recall.

3.4.4. Tools and Environment

The training and testing were conducted in the following hardware and software environments:

- GPU: NVIDIA A100 (80 GB)
- Processor: Intel Xeon Platinum 8358 CPU 2,60 GHz, 6 cores
- RAM : 32 GB
- Operating System: Ubuntu 22.04.4 LTS
- Framework : PyTorch[8], Conda[9]

The 80 GB memory of the NVIDIA A100 GPU allows efficient handling of long code sequences while maintaining reasonable inference times (about 30 minutes per epoch).

3.5. RESULTS

Tables 3.6 and 3.8 present the results of LOPOCV. Each row i shows the test performances on i^{th} project of the model trained on all projects excluding i.

3.5.1. Token-Contextless Results

Table 3.6 and 3.7 presents the overall results obtained during the experiment without the project token-contexts information. The DeepSeek-Coder model achieved an average AUC of 0.89 and an average F1-score of 0.80 across all tested projects, indicating a generally

[8] https://pytorch.org
[9] https://anaconda.org/anaconda/conda

correct ability to distinguish between defective and non-defective classes. However, significant variability was observed across different projects.

The analysis of results by project highlights heterogeneous performance:

- Velocity and Log4j achieved the best results with F1-scores above 0.95 and AUC scores close to 0.98.
- Camel produced the weakest results, with an F1-score of 0.65.
- All other projects showed intermediate results, with F1-scores between 0.72 and 0.82, indicating a good generalization capability of the model.

The confusion matrices (Table 3.7) show that the model tends to generate a slightly higher number of false negatives in large projects such as Camel and Xalan, which directly affect recall. Conversely, smaller projects such as Velocity show high precision and recall, suggesting that the model can better identify defect-leading patterns in less complex projects.

The AUC values (Table 3.6) confirm this trend. The high AUC scores for Velocity and Log4j indicate a strong discriminative capacity of the model, with a clear separation between defective and non-defective classes. In contrast, the lower AUC score for Camel reflects the model's difficulty in distinguishing between classes in this project.

Table 3.6 : Scores of token-contextless models

Projects	**Precisions**	**Recalls**	**F1-scores**	**AUC**
Velocity	0.91	0.98	0.94	0.98
Log4j	0.93	0.97	0.95	0.98
Camel	0.66	0.63	0.65	0.75
Xalan	0.72	0.71	0.72	0.82
POI	0.80	0.79	0.79	0.88
Synapse	0.81	0.80	0.80	0.89
JEdit	0.82	0.80	0.81	0.91
Xerces	0.82	0.75	0.78	0.89
Lucene	0.83	0.77	0.80	0.90
Ant	0.83	0.74	0.78	0.89

Table 3.7 : Confusion matrix of token-contextless models

Projects	TN	FP	FN	TP
Velocity	85	22	4	237
Log4j	151	30	11	403
Camel	703	275	308	546
Xalan	1100	357	381	957
Poi	1246	371	397	1531
Synapse	1459	383	408	1660
Jedit	1983	406	442	1873
Xerces	2025	412	608	1904
Lucene	2113	436	643	2225
Ant	2698	482	822	2358

3.5.1.1. Detailed Analysis

The detailed project-by-project analysis highlights significant performance gaps:

- Best Performances: The Velocity and Log4j projects present the best results in terms of F1-score (0.94 for Velocity, 0.95 for Log4j) and AUC (0.98 and 0.98, respectively). Both projects are relatively small (# of classes) and present a moderately balanced class distribution (non-defective/defective ratio below 1), which facilitates model generalization.
- Weaker Performances: The Camel project produced significantly lower performance (AUC of 0.75), likely due to its large size (1648 classes) and a marked initial imbalance between defective and non-defective classes (ratio of 2.68).

This contextless increases the model's difficulty in generalizing effectively.

3.5.1.2. Impact of Training Dataset specificities

The analysis of the results suggests a relationship between class distribution of training datasets and the model's generalization capability:

- Large Projects (Camel, Ant): When these projects are used for training, the model shows a reduced ability to generalize. This difficulty could be explained by the increased complexity of code structures and internal dependencies in large projects.
- Small Projects (Velocity, Log4j): These projects lead to more robust generalizations. Their smaller size and moderately balanced distribution seem to promote more efficient learning of bug patterns.

The above results highlight the model's difficulty in effectively generalizing when the specific project context is missing. Adding project token is intended to provide additional contextual information, allowing the model to better capture project-specific variations and improve generalizations, especially for large or imbalanced projects.

3.5.2. Token-Context Results

Table 3.8 and 3.9 presents the results after introducing the project token-contexts. The overall AUC decreased slightly to 0.81, while recall values increased significantly.

We note that the improvement is particularly notable for Camel (+15% recall) and Ant (+6.42%), confirming that the token allows the model to better capture the specific characteristics of these challenging projects. The performance is more stable for smaller projects.

Table 3.8 : Scores of tokens-context models

Projects	**Precisions**	**Recalls**	**F1-scores**	**AUC**
Velocity	0.90	0.9	0.93	0.96
Log4j	0.92	0.96	0.94	0.97
Camel	0.55	0.78	0.64	0.74
Xalan	0.58	0.68	0.63	0.71
POI	0.69	0.7	0.7	0.81
Synapse	0.7	0.77	0.74	0.83
JEdit	0.72	0.78	0.75	0.86
Xerces	0.74	0.80	0.77	0.8
Lucene	0.76	0.82	0.78	0.88
Ant	0.72	0.80	0.76	0.86

Table 3.9 : Confusion matrix of token-context models

Projects	**TN**	**FP**	**FN**	**TP**
Velocity	82	25	7	234
Log4j	150	31	15	399
Camel	442	536	185	669
Xalan	820	637	428	910
Poi	959	658	437	1491
Synapse	1179	663	456	1612
Jedit	1685	704	495	1820
Xerces	1733	704	495	2017
Lucene	1810	739	515	2353
Ant	2197	983	618	2562

3.5.2.1. Detailed Analysis

A detailed project-by-project analysis after introducing the project token-contexts provides a clearer understanding of its impact on the model's generalization capability.

Including token-contexts leads to a significant increase of recall values for the projects Camel (+15%), Ant (+6.42%) and Xerces (+4.88%) as shown in Tables 3.8 and 3.10. The improvement confirms that the model benefits from better contextualization, allowing it to better capture the structural and internal logic of the code in these more complex and imbalanced projects.

We also noted that with project token-contexts, Ant AUC score decreases (-3.5%), when model performances on Xalan remained relatively stable (F1-score from 0.72 to 0.63), suggesting that the model already captures sufficient patterns specific to the code without needing additional context.

The model already performs well on Velocity and Log4j systems without token-context. The F1-score remain relatively unchanged, with a slight decrease (-1.27%) on Velocity, and (-0.64%) on Log4j. This may be explained by the well balanced and small size of both projects as test dataset, allowing the model to well generalize without additional context.

The confusion matrices (Table 3.9) indicate a notable reduction in the number of false negatives in the projects, confirming the positive effect of contextualization.

3.5.2.2. Impact of Project specificity on Results With Token

An in-depth analysis of the results reveals a systematic trend between project size and the effect of including the token-context:

- Large Projects: Camel and Ant clearly benefit from adding the token, with an average recall improvement of +9.38%. This improvement confirms that the model leverages contextual information in complex and imbalanced projects, better capturing the underlying code structure.
- Small Projects: Velocity and Log4j show little change after adding the token. The small improvement (less than 1% variation in recall) is explained by the model's

initial ability to generalize effectively due to simpler code structures and balanced distribution.

- Moderately Imbalanced Projects: For Xalan, POI, and Lucene, including the token-contexts produced no significant effect or even a slight degradation in results. This may be due to an insufficient level of complexity for the model to benefit from additional context.

3.5.2.3. In-Depth Comparison of token context impact

Table 3.9 summarizes the results obtained with and without the token, comparing the F1-score, recall, and AUC for each tested project.

Key Findings:

- Velocity and Log4j recorded minor decreases in F1-score and AUC after adding the token, which suggests that the model already generalizes well for these balanced, small-scale projects without needing additional context.
- The largest improvement was seen in Camel (+15%), where the recall increased from 0.63 to 0.78. This confirms that adding contextual information helps the model to better detect defective classes in complex and highly imbalanced projects. Ant (+6.42%), and Xerces (+4.88%) also showed significant improvement in recall, reinforcing the positive effect of contextualization in large or imbalanced projects.
- For Xalan, POI, and Jedit, the addition of the token resulted in a decrease in F1-score and AUC, suggesting that the model may already be capturing sufficient contextual information without needing explicit project context.
- The decrease in AUC for Xalan (-13.08%) and POI (-8.00%) indicates that adding contextual information increases sensitivity to false positives, reducing the overall discriminative power of the model.

•

Table 3.10 : % of Performance variation Δ

Projects	Δ AUC (%)	%Δ F1 (%)	Δ Recall (%)
Velocity	-1.58	-1.27	-1.24
Log4j	-1.13	-0.64	-0.96
Camel	-0.73	-3.23	+15
Xalan	-13.08	-12.61	-4.9
POI	-8.00	-8.51	-2.62
Synapse	-6.82	-6,60	-0.03
JEdit	-5.7	-7.37	-2.83
Lucene	-2.49	-1.90	0.00
Ant	-3.73	-1,59	+6,42
Xerces	-2.25	-2.27	+4.88

3.5.2.4. Performance Trade-Off

While adding the token improved the model's sensitivity to detecting defective classes (higher recall), it also increased the false positive rate, reducing precision and AUC. This trade-off is expected since increasing sensitivity typically leads to more false positives, especially in complex and imbalanced datasets.

However, the increase in recall is highly beneficial for large and complex projects where identifying defective classes is more challenging. The improved recall observed in Camel, Ant, and Xerces indicates that the model including project-specific contextual information is more effective.

3.5.2.5. Contextual Sensitivity

The contrasting effect of adding the token across different projects highlights the importance of project size and class distribution:

- In large, complex projects such as Camel and Ant, adding contextual information allowed the model to better identify defective classes.
- In smaller or balanced projects such as Velocity and Log4j, adding the token had minimal impact since the model already performed well without project-specific context.

- In moderately imbalanced projects like Xalan and POI, the additional context increased sensitivity to false positives, reducing overall discriminative performance.

3.5.2.6. Generalization and Scalability

The ability of the model to generalize across different projects remains a key challenge in software defects prediction. While adding the token improved recall and contextual sensitivity in some cases, the decrease in AUC score for several projects suggests that the model's sensitivity to project-specific variations introduces new complexities. The balance between precision and recall remains crucial for deploying the model in real-world production environments. High recall ensures that most defects are detected, but lower precision increases the risk of false positives, which could impact development efficiency.

These findings suggest that the optimal use of project-specific context should be adaptive:

- Adding the token is beneficial for large and imbalanced projects.
- For smaller and well-balanced projects, the model generalizes effectively without additional contextual information.

While traditional models, as Multilayer Perceptron (MLP), CNNs, SVMs, have been extensively evaluated on PROMISE datasets, their inputs typically rely on handcrafted metrics rather than raw source code. Due to differences in data preprocessing, metric selection, and evaluation protocols, a direct numerical comparison would be misleading. Instead, our study focuses on demonstrating the feasibility and effectiveness of end-to-end source code modeling using LLMs.

3.6. DISCUSSIONS

3.6.1. Model Performance

The project-specific results of the model allow us to identify several factors that explain the variability in performance.

DeepSeek-Coder benefits from the Rotary Position Embeddings (RoPE) and Grouped-Query Attention (GQA) mechanisms, which help to better capture long-range dependencies in the code.

3.6.1.1. Influence of Project Size

The analysis by project size (Table 3.6) shows that small projects generally display better performance in terms of F1-score and AUC compared to larger projects.

Small projects (e.g., Velocity, Log4j) have an average F1-score that is 15% higher than large projects (e.g., Camel, Xalan). These observations may be explained by the following factors:

- Better balance – Small projects are generally more balanced in terms of defectgy/non-defectgy class distribution.
- Lower complexity – The reduced complexity of small projects facilitates the learning of underlying patterns.
- Higher structural variability – Large projects introduce greater structural variability, which complicates generalization.

Table 3.11 : Performance comparison based on project size

Category	**Projects**	**Avg.F1-score**	**Avg. AUC**
Small Projects	Velocity, Log4j, Xerces, Synapse	0.90	0.94
Large Projects	Camel, Xalan, POI, Lucene, Ant	0.74	0.85

3.6.1.2. Influence of Class Distribution

The imbalance in the defectgy/non-defectgy class distribution also has a significant impact.

In the Camel project (ratio of 2.68), the model tends to favor the majority class (non-defectgy), leading to a drop in recall.

Conversely, in the Log4j project (ratio of 0.61), the small proportion of non-defectgy classes allows the model to improve its predictive capacity, resulting in increased recall.

This behavior is confirmed by the comparison of confusion matrices:

- The model generates more false negatives in highly imbalanced projects (Camel, Xalan).
- Balanced projects (Velocity, Log4j) exhibit a more homogeneous distribution of predictions.

3.6.2. Limiting Factors

The results indicate that DeepSeek-Coder faces certain specific challenges when predicting software defects:

- Logical complexity – Large projects (e.g., Camel) introduce high structural variability, which complicates the learning of recurring patterns.
- Class imbalance – Highly imbalanced projects (e.g., Camel, Xalan) exhibit an increase in false negatives, suggesting that the model tends to favor the majority class.
- Size bias – Large projects require higher memory and processing capacity, which may limit the model's efficiency in large-scale scenarios.

These observations highlight the need to introduce additional contextual information to improve the model's generalization ability.

3.6.2.1. Impact of the contextual Token

3.6.2.1.1. Defect/Non-Defect Ratio

Initially highly imbalanced projects, such as Camel (initial ratio of 2.68) and Ant (ratio of 2.98), show a significant improvement in the model’s ability to predict defective classes after rebalancing, with an increase in recall of +15% and +6.42% respectively, following the addition of the token.

On the other hand, for already well-balanced projects, such as Velocity (initial ratio of 0.69) and Log4j (initial ratio of 0.61), the effect is negative, indicating that the model can learn effectively in balanced contexts without requiring additional contextual signals.

Table 3.12 : Impact of defect ratio in model performance

Projects	**Initial Ratio**	**Difference (%)**
Camel	2.68	+15%
Ant	2.98	+6.42%
Velocity	0.69	-1.24%
Log4j	0.61	-0.96%

3.6.2.1.2. Impact of Test Project Size

The analysis of the results shows that the size of the project under test can significantly impact the model's performance:

- When testing model against large projects (e.g., Camel, Xalan), it tends to achieve lower AUC and precision probably due to the increased complexity of defect-leading patterns in large and heterogeneous projects.
- Conversely, when tested against small projects (e.g., Velocity, Log4j), the model maintains high performance, reflecting a strong ability to capture code structure in simple and well-defined contexts.

The positive effect of the token is more noticeable when the test project is complex, confirming that the model particularly benefits from contextual information in these cases.

Table 3.13 : Impact of test dataset size on model performance

Categories	**Projects**	**Average values of**		
		Δ Recall	**Δ F1**	**Δ AUC**
Large	Camel, Ant, Xalan	+9.38%	+6.42%	-3.42%
Small	Velocity, Log4j	-0.96%	-1.27%	-1.13%
Medium	Jedit, Lucene	-0.75%	-1.38%	-1.10%

3.6.2.2. Contribution and Limitations of the token-context

The most notable effect of adding the token is noted in the recalls scores, particularly for large and imbalanced projects such as Camel and Ant. The improvement reflects the model's improved ability to predict defective classes in complex contexts.

Although the model was not trained on Camel, the token-contexts seem to enable the transfer of structural patterns learned in other projects related to syntactic and semantic similarities:

- Knowledge transfer: Similar code structures between Camel and Ant (e.g., exception handling) are recognized through the project token-context.
- Global context impact : The project token-context acts as an anchor, aligning the model’s behavior with familiar patterns observed in other projects.

3.6.2.3. Generalization Factors

The LOPOCV highlights several key factors impacting the model’s ability to generalize across projects:

- Test project size – Larger and more complex projects (e.g., Camel, Ant, Xalan) tend to reduce the model's generalization ability probably due to the diversity and complexity of patterns to be captured.
- Divergence between projects – Even with the token-context, the model struggles to generalize when the source code structure and patterns significantly differ between the training and test projects.

3.6.2.4. Future Directions

The experimental results have revealed both the strengths and limitations of DeepSeek-Coder in the context of software defects prediction. To strengthen the model's generalization capability and improve the robustness of predictions, several improvements are possible:

- Dynamic adaptation of the project token-contexts: Activating or deactivating the token-context based on the degree of imbalance detected in the target project could improve generalization. A meta-model could automatically adjust the token's activation based on the characteristics of the project, maximizing recall without degrading precision. This approach would allow the model to dynamically adapt to projects with varying levels of imbalance, such as Camel or Ant.
- Dynamically determining the optimal classification threshold for each project during cross-validation could improve recall and F1-score. An adjusted threshold based on class imbalance would allow better defect detection in projects such as Camel without compromising overall precision. The approach would help the model to adapt to distribution variations across projects.
- Adopting a larger variant (e.g., DeepSeek-Coder 7B) would increase the model's ability to capture complex relationships in the code. A larger context window and enhanced architecture could improve precision and recall scores, particularly for long and complex projects. This approach would enhance the model's ability to process long sequences and detect complex code structures.
- Pre-training the model on a multi-project corpus, followed by fine-tuning with the project token-context for each project, could improve generalization. This strategy

would allow the model to capture both general and project-specific patterns, increasing the model's ability to adapt to new projects with different code structures.

3.7. CONCLUSION

The current work presented a fine-tuned transformer-based approach for software defects prediction that bypasses the use of proxy artifacts such as metrics or graph-based representations. Our model uses the DeepSeek-Coder backbone, fine-tuned for sequence classification, to analyze raw source code, preserving its syntactic, semantic, and contextual richness. We introduced a novel strategy of embedding project-specific contextual tokens, allowing the model to adapt to varying project structures and coding styles, and significantly improving generalization capabilities.

The extensive evaluation using the leave one project out cross- validation on ten diverse Java open-source projects shows the model's strong performance, achieving 0.93 of precision, 0.97 of recall, 0.95 of F1-score, and 0.98 of AUC. The approach effectively addresses long-standing challenges in defect prediction, including information loss, preprocessing overhead, and limited cross-project scalability.

By removing the need for intermediate artifacts our model paves the way for more efficient, scalable, and real-world deployable defect prediction tools. Future work will explore dynamic contextualization strategies, threshold optimization, and scaling the model further using larger DeepSeek-Coder variants to enhance its applicability in large-scale industrial environments.

3.8. REFERENCES

[1] Q. Song, Y. Guo, and M. Shepperd, "A comprehensive investigation of the role of imbalanced learning for software defect prediction," IEEE Transactions on Software Engineering, vol. 45, no. 12, pp. 1253-1269, 2018.

[2] K. Huckvale, S. Adomaviciute, J. T. Prieto, M. K.-S. Leow, and J. Car, "Smartphone apps for calculating insulin dose: a systematic assessment," BMC medicine, vol. 13, pp. 1-10, 2015.

[3] K. Lentine et al., "OPTN/SRTR 2020 annual data report: kidney," American journal of transplantation, vol. 22, pp. 21-136, 2022.

[4] C. Saltapidas and R. Maghsood, "Financial risk the fall of knight capital group," University of Gothenburg, pp. 1-8, 2018.

[5] K. H. LaCommare and J. H. Eto, "Understanding the cost of power interruptions to US electricity consumers," Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), 2004.

[6] D. H. O'Dell, "The Debugging Mindset: Understanding the psychology of learning strategies leads to effective problem-solving skills," Queue, vol. 15, no. 1, pp. 71-90, 2017.

[7] J. M. Reddy, K. Muthukumaran, H. Shahriar, V. Clincy, and N. Sakib, "Comprehensive Feature Extraction for Cross-Project Software Defect Prediction," in 2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC), 2022: IEEE, pp. 450-451.

[8] Q. Dou, D. Coelho de Castro, K. Kamnitsas, and B. Glocker, "Domain generalization via model-agnostic learning of semantic features," Advances in neural information processing systems, vol. 32, 2019.

[9] M. Scotto, A. Sillitti, G. Succi, and T. Vernazza, "A relational approach to software metrics," in Proceedings of the 2004 ACM symposium on Applied computing, 2004, pp. 1536-1540.

[10] J. Li, P. He, J. Zhu, and M. R. Lyu, "Software defect prediction via convolutional neural network," in 2017 IEEE international conference on software quality, reliability and security (QRS), 2017: IEEE, pp. 318-328.

[11] Z. Li, X.-Y. Jing, and X. Zhu, "Progress on approaches to software defect prediction," Iet Software, vol. 12, no. 3, pp. 161-175, 2018.

[12] P. J. Lisboa, "Interpretability in machine learning–principles and practice," in International Workshop on Fuzzy Logic and Applications, 2013: Springer, pp. 15-21.

[13] D. Guo et al., "DeepSeek-Coder: When the Large Language Model Meets Programming--The Rise of Code Intelligence," arXiv preprint arXiv:2401.14196, 2024.

[14] A. Vaswani et al., "Attention is all you need," Advances in neural information processing systems, vol. 30, 2017.

[15] Y. Yu et al., "Fine-tuning large language models to improve accuracy and comprehensibility of automated code review," ACM transactions on software engineering and methodology, vol. 34, no. 1, pp. 1-26, 2024.

[16] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," IEEE Transactions on software engineering, vol. 20, no. 6, pp. 476-493, 1994.

[17] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," IEEE Transactions on software engineering, vol. 28, no. 1, pp. 4-17, 2002.

[18] N. E. Fenton and M. Neil, "Software metrics: successes, failures and new directions," Journal of Systems and Software, vol. 47, no. 2-3, pp. 149-157, 1999.

[19] B. Curtis, S. B. Sheppard, P. Milliman, M. Borst, and T. Love, "Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics," IEEE Transactions on software engineering, no. 2, pp. 96-104, 1979.

[20] S. Karim, H. L. H. S. Warnars, F. L. Gaol, E. Abdurachman, and B. Soewito, "Software metrics for fault prediction using machine learning approaches: A literature review with PROMISE repository dataset," in 2017 IEEE international conference on cybernetics and computational intelligence (CyberneticsCom), 2017: IEEE, pp. 19-23.

[21] A. Alsaeedi and M. Z. Khan, "Software defect prediction using supervised machine learning and ensemble techniques: a comparative study," Journal of Software Engineering and Applications, vol. 12, no. 5, pp. 85-100, 2019.

[22] M. E. Aydin, "Leveraging Deep Learning for Enhanced Software Fault Prediction Using Error-type Metrics."

[23] T. Siddiqui and M. Mustaqeem, "Performance evaluation of software defect prediction with NASA dataset using machine learning techniques," International Journal of Information Technology, vol. 15, no. 8, pp. 4131-4139, 2023.

[24] G. Esteves, E. Figueiredo, A. Veloso, M. Viggiato, and N. Ziviani, "Understanding machine learning software defect predictions," Automated Software Engineering, vol. 27, no. 3, pp. 369-392, 2020.

[25] F. Subhan, X. Wu, L. Bo, X. Sun, and M. Rahman, "A deep learning-based approach for software vulnerability detection using code metrics," IET software, vol. 16, no. 5, pp. 516-526, 2022.

[26] H. Liang, Y. Yu, L. Jiang, and Z. Xie, "Seml: A semantic LSTM model for software defect prediction," IEEe Access, vol. 7, pp. 83812-83824, 2019.

[27] C. Pan, M. Lu, B. Xu, and H. Gao, "An improved CNN model for within-project software defect prediction," Applied Sciences, vol. 9, no. 10, p. 2138, 2019.

[28] J. Chen et al., "Software visualization and deep transfer learning for effective software defect prediction," in Proceedings of the ACM/IEEE 42nd international conference on software engineering, 2020, pp. 578-589.

[29] A. V. Phan, M. Le Nguyen, and L. T. Bui, "Convolutional neural networks over control flow graphs for software defect prediction," in 2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI), 2017: IEEE, pp. 45-52.

[30] C. Pan, M. Lu, and B. Xu, "An empirical study on software defect prediction using codebert model," Applied Sciences, vol. 11, no. 11, p. 4793, 2021.

[31] M. N. Uddin, B. Li, Z. Ali, P. Kefalas, I. Khan, and I. Zada, "Software defect prediction employing BiLSTM and BERT-based semantic feature," Soft Computing, vol. 26, no. 16, pp. 7877-7891, 2022.

[32] A. Dal Pozzolo, O. Caelen, and G. Bontempi, "When is undersampling effective in unbalanced classification tasks?," in Joint european conference on machine learning and knowledge discovery in databases, 2015: Springer, pp. 200-215.

[33] A. W. C. Faria, C. L. de Castro, and A. de Pádua Braga, "A New Oversampling-Based Approach for Class Imbalance Problem," 2016.

[34] C. Elkan, "The foundations of cost-sensitive learning," in International joint conference on artificial intelligence, 2001, vol. 17, no. 1: Lawrence Erlbaum Associates Ltd, pp. 973-978.

[35] A. Okutan and O. T. Yıldız, "Software defect prediction using Bayesian networks," Empirical Software Engineering, vol. 19, pp. 154-181, 2014.

[36] J. Gonçalves et al., "Evaluating LLaMA 3.2 for Software Vulnerability Detection," arXiv preprint arXiv:2503.07770, 2025.

[37] Z. Feng et al., "Codebert: A pre-trained model for programming and natural languages," arXiv preprint arXiv:2002.08155, 2020.

[38] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," arXiv preprint arXiv:2103.06333, 2021.

[39] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," arXiv preprint arXiv:2109.00859, 2021.

[40] G. Boetticher, T. Menzies, and T. Ostrand, "PROMISE Repository of Empirical Software Engineering Data, West Virginia University, Department of Computer Science, 2007," ed.

[41] L. Tang, C. Tao, H. Guo, and J. Zhang, "Software defect prediction via gcn based on structural and context information," in 2022 9th International Conference on Dependable Systems and Their Applications (DSA), 2022: IEEE, pp. 310-319.

[42] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in 2013 35th international conference on software engineering (ICSE), 2013: IEEE, pp. 382-391.

[43] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in Proceedings of the 30th international conference on Software engineering, 2008, pp. 181-190.

[44] K. O. Elish and M. O. Elish, "Predicting defect-prone software modules using support vector machines," Journal of Systems and Software, vol. 81, no. 5, pp. 649-660, 2008.

[45] N. Nagappan and T. Ball, "Using software dependencies and churn metrics to predict field failures: An empirical case study," in First international symposium on empirical software engineering and measurement (ESEM 2007), 2007: IEEE, pp. 364-373.

[46] X.-Y. Jing, S. Ying, Z.-W. Zhang, S.-S. Wu, and J. Liu, "Dictionary learning based software defect prediction," in Proceedings of the 36th international conference on software engineering, 2014, pp. 414-423.

[47] S. Motogna, D. Lupsa, and I. Ciuciu, "A NLP approach to software quality models evaluation," in OTM Confederated International Conferences" On the Move to Meaningful Internet Systems", 2018: Springer, pp. 207-217.

[48] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "Automated parameter optimization of classification techniques for defect prediction models," in Proceedings of the 38th international conference on software engineering, 2016, pp. 321-332.

[49] A. LeClair, S. Haque, L. Wu, and C. McMillan, "Improved code summarization via a graph neural network," in Proceedings of the 28th international conference on program comprehension, 2020, pp. 184-195.

[50] H. He and E. A. Garcia, "Learning from imbalanced data," IEEE Transactions on knowledge and data engineering, vol. 21, no. 9, pp. 1263-1284, 2009.

[51] P. Branco, L. Torgo, and R. P. Ribeiro, "A survey of predictive modeling on imbalanced domains," ACM computing surveys (CSUR), vol. 49, no. 2, pp. 1-50, 2016.

[52] N. V. Chawla, "Data mining for imbalanced datasets: An overview," Data mining and knowledge discovery handbook, pp. 875-886, 2010.

[53] J. Su, M. Ahmed, Y. Lu, S. Pan, W. Bo, and Y. Liu, "Roformer: Enhanced transformer with rotary position embedding," Neurocomputing, vol. 568, p. 127063, 2024.

[54] J. Ainslie, J. Lee-Thorp, M. De Jong, Y. Zemlyanskiy, F. Lebrón, and S. Sanghai, "Gqa: Training generalized multi-query transformer models from multi-head checkpoints," arXiv preprint arXiv:2305.13245, 2023.

[55] H. Touvron et al., "Llama: Open and efficient foundation language models," arXiv preprint arXiv:2302.13971, 2023.

[56] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," OpenAI blog, vol. 1, no. 8, p. 9, 2019.

[57] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," arXiv preprint arXiv:1711.05101, 2017.

[58] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout : a simple way to prevent neural networks from overfitting," The journal of machine learning research, vol. 15, no. 1, pp. 1929-1958, 2014.

[59] I. Beltagy, K. Lo, and A. Cohan, "SciBERT: A pretrained language model for scientific text," arXiv preprint arXiv:1903.10676, 2019.

[60] I. Loshchilov and F. Hutter, "Sgdr: Stochastic gradient descent with warm restarts," arXiv preprint arXiv:1608.03983, 2016.

[61] P. Micikevicius et al., "Mixed precision training," arXiv preprint arXiv:1710.03740, 2017.

[62] S. Ruder, "An overview of gradient descent optimization algorithms," arXiv preprint arXiv:1609.04747, 2016.

CHAPITRE 4
DISCUSSION GÉNÉRALE

Ce mémoire s'inscrit dans une approche exploratoire de la prédiction des défauts logiciels directement à partir du code source brut, sans recourir à des représentations intermédiaires telles que les métriques logicielles ou les graphes structurels. L'objectif consiste à exploiter les régularités syntaxiques, sémantiques et structurelles présentes dans le code, en tirant parti du potentiel des LLMs. Après une phase d'expérimentation impliquant plusieurs modèles entraînés et évalués sur la base PROMISE selon la méthodologie détaillée dans l'article (Chapitre 3), le modèle DeepSeek-Coder a été retenu pour l'étude principale grâce à sa spécialisation et à ses performances supérieures.

4.1. Limites des modèles BigBird et StarCoder

Les expérimentations préliminaires avec BigBird et StarCoder ont permis d'identifier clairement certaines limites.

BigBird, malgré sa capacité à traiter des séquences très longues grâce à son mécanisme d'attention clairsemée, repose sur un pré-entraînement orienté texte naturel. Par conséquent, il ne dispose pas des inducteurs structurels adéquats pour interpréter précisément les constructions logiques du code source, ce qui limite fortement sa capacité à identifier des motifs défectueux dans les programmes. Ces contraintes se traduisent par un faible F1-score moyen de 0,42 et une AUC moyenne de 0,54.

StarCoder bénéficie d'un pré-entraînement sur un corpus massif et diversifié de code. Cependant, son entraînement initial ne cible pas explicitement la classification binaire selon la présence ou non de défauts. Il reste donc peu sensible aux régularités spécifiques menant à l'apparition de défauts logiciels. Cela se reflète dans ses performances, avec un F1-score moyen de 0,56 et une AUC moyenne de 0,63.

4.2. Intérêts et Performances de DeepSeek-Coder

Contrairement aux modèles précédents, DeepSeek-Coder s'est avéré particulièrement performant pour la prédiction de défauts logiciels (Tableau 4.1). Son pré-entraînement sur des dépôts de code à grande échelle lui confère une capacité à capter les dépendances

sémantiques et structurelles complexes. De plus, ses mécanismes d'encodage positionnel Rotary (RoPE) et d'attention groupée (GQA) permettent une meilleure prise en compte des relations à longue portée dans le code source.

Grâce à sa fenêtre contextuelle étendue (jusqu'à 16 000 tokens), le modèle peut analyser l'intégralité d'une classe Java, captant ainsi les motifs logiques et les dépendances qui échappent aux modèles plus restreints.

Les résultats expérimentaux confirment cette supériorité, avec un F1-score moyen de 0,80 et une AUC de 0,89 sans contexte, et jusqu'à 0,95/0,98 respectivement dans certaines configurations. L'ajout de tokens contextuels renforce la robustesse du modèle dans les scénarios inter-projets, avec une amélioration du rappel notable sur certains projets tels que Camel (+14,4 %), Xerces (+4,88 %) et Ant (+6,42 %).

Tableau 4.1 : Performances des modèles utilisés

Modèles	**Valeurs moyennes de**	
	F1	**AUC**
BigBird	0,42	0,54
StarCoder	0,56	0,63
DeepSeek-Coder	**0,80**	**0,89**

4.3. Portée, Limites et Facteurs de variabilité

Malgré ces résultats prometteurs, plusieurs limitations doivent être soulignées :

- Pré-entraînement non supervisé : DeepSeek-Coder n'a pas été pré-entraîné sur des annotations de défauts. Cela limite sa capacité à identifier certains défauts complexes ou rares. Un pré-entraînement orienté bug detection pourrait améliorer ses performances.
- Effets contrastés du token contextuel : Bien que bénéfique pour les projets volumineux et déséquilibrés, l'introduction de tokens contextuels peut dégrader les performances dans les projets équilibrés ou de petite taille (ex. : Xalan), avec une diminution du F1-score (-12,61 %) et de l'AUC (-13,08 %).
- Facteurs de généralisation : La taille du projet, la complexité structurale et le déséquilibre entre classes influencent fortement la capacité de généralisation du

modèle. Les grands projets comme Camel ou Xalan réduisent l'efficacité du modèle par effet de surcharge ou de sur-ajustement.

- Validité externe : L'approche a été validée exclusivement sur des projets Java. Son adaptation à d'autres langages ou à d'autres types de défauts (vulnérabilités, problèmes de performance) reste à explorer.

4.4. Perspectives

Plusieurs pistes de recherche pourraient prolonger ces travaux :

- Pré-entraînement sur des données annotées : Enrichir le modèle avec des corpus contenant des étiquettes de défauts explicites permettrait d'optimiser ses capacités de classification.
- Contextualisation adaptative : Activer dynamiquement les tokens contextuels en fonction des caractéristiques du projet (déséquilibre, taille, complexité) permettrait d'améliorer la précision sans sacrifier le rappel.
- Optimisation du seuil de classification : Ajuster le seuil de décision de manière projet-spécifique pourrait corriger le biais vers la classe majoritaire dans certains cas.
- Extension à d'autres langages : Tester l'approche sur des projets Python, C++ ou JavaScript permettrait d'évaluer sa généricité.

Détection fine de défauts : Une classification multi-label visant à identifier le type de défaut (bogue logique, anomalie de performance, faille de sécurité) constituerait une avancée significative.

Ces résultats ouvrent la voie à une nouvelle génération de modèles prédictifs capables d'analyser le code source dans toute sa complexité, en capturant directement les caractéristiques menant aux anomalies logicielles, sans dépendre d'artéfacts intermédiaires. L'approche directe incarnée par DeepSeek-Coder représente une avancée concrète vers une assurance qualité logicielle plus efficace, automatisée et généralisable.

CHAPITRE 5
CONCLUSION

Ce mémoire s'inscrit dans le domaine de la prédiction des défauts logiciels par analyse directe du code source brut, en exploitant le potentiel des LLMs pour détecter automatiquement les classes susceptibles d'être défectueuses. L'objectif était de dépasser les approches traditionnelles basées sur des artefacts intermédiaires, en mobilisant la capacité des LLMs à saisir directement la structure et les anomalies du code.

Lors d'une première phase exploratoire, plusieurs modèles ont été évalués, notamment BigBird et StarCoder, permettant de mettre en évidence leurs limites en termes de spécialisation et de sensibilité aux défauts. Ces observations ont conduit à privilégier DeepSeek-Coder, un modèle pré-entraîné spécifiquement sur des projets logiciels et doté d'une fenêtre contextuelle étendue.

Les résultats obtenus confirment clairement la pertinence de cette approche directe. DeepSeek-Coder a démontré sa capacité à analyser efficacement des classes complètes et à généraliser entre des projets variés. Les performances observées sur les jeux de données issus de PROMISE, avec un F1-score moyen et une AUC moyenne élevés (respectivement 0.80 et 0.89), attestent de l'intérêt et de la faisabilité de cette méthode.

Cependant, cette recherche présente certaines limites : absence d'un pré-entraînement supervisé sur des défauts annotés, activation uniforme des tokens contextuels, seuil fixe de classification, ainsi qu'une évaluation limitée au langage Java. Ces aspects ouvrent des perspectives intéressantes pour des recherches futures, telles qu'un affinement supervisé du modèle à partir d'annotations réelles, une gestion dynamique des entrées en fonction des caractéristiques des projets, ou encore une extension à d'autres langages et catégories de défauts logiciels.

D'un point de vue plus global, ce travail révèle le potentiel des LLMs spécialisés pour améliorer la qualité logicielle, en proposant des outils prédictifs plus performants, mieux adaptés à une intégration dans les environnements réels de développement. Il contribue ainsi à poser les fondements d'une nouvelle génération de méthodes plus expressives, exhaustives et potentiellement plus robustes.

Enfin, en se positionnant au carrefour de la recherche académique et des exigences industrielles, ce mémoire favorise le rapprochement concret entre l'intelligence artificielle et les pratiques actuelles d'assurance qualité logicielle.

RÉFÉRENCES

[1] D. H. O'Dell, « The Debugging Mindset: Understanding the psychology of learning strategies leads to effective problem-solving skills, » Queue, vol. 15, no. 1, pp. 71-90, 2017.

[2] G. Abaei, M. R. Mashinchi, and A. Selamat, "Software fault prediction using BP-based crisp artificial neural networks," International Journal of Intelligent Information and Database Systems, vol. 9, no. 1, pp. 15-31, 2015.

[3] A. Vaswani et al., "Attention is all you need," Advances in neural information processing systems, vol. 30, 2017.

[4] M. O. Normalizacyjna, Systems and Software Engineering-Systems and Software Quality requirements and evaluation (SQuaRE)-System and Software Quality models. ISO, 2011.

[5] T. Hynninen, J. Kasurinen, and O. Taipale, "Framework for observing the maintenance needs, runtime metrics and the overall quality-in-use," Journal of Software Engineering and Applications, vol. 11, no. 4, pp. 139-152, 2018.

[6] K. H. Bennett and V. T. Rajlich, "Software maintenance and evolution: a roadmap," in Proceedings of the Conference on the Future of Software Engineering, 2000, pp. 73-87.

[7] D. Russo, "Socio–Technical Software Engineering : a Quality–Architecture–Process Perspective," 2019.

[8] A. I. M. Klaura, "SCRAP-Secure Code Review Automated Platform. Prototype of an automated feed-back platform to provide secure coding feed-back for introductory programming courses," 2020.

[9] P. Kruchten and I. Ozkaya, Managing technical debt: reducing friction in software development. Addison-Wesley Professional, 2019.

[10] J. Nielsen, Usability engineering. Morgan Kaufmann, 1994.

[11] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," IEEE Transactions on software engineering, vol. 20, no. 6, pp. 476-493, 1994.

[12] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," IEEE Transactions on software engineering, vol. 28, no. 1, pp. 4-17, 2002.

[13] T. Gyimóthy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," IEEE Transactions on Software engineering, vol. 31, no. 10, pp. 897-910, 2005.

[14] B. Curtis, S. B. Sheppard, P. Milliman, M. Borst, and T. Love, "Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics," IEEE Transactions on software engineering, no. 2, pp. 96-104, 1979.

[15] R. Harper, "Programming languages: Theory and practice," Course Notes on the WWW, 2002.

[16] M. Weiser, « Program slicing, » IEEE Transactions on software engineering, no. 4, pp. 352-357, 2009.

[17] K. J. Ottenstein and L. M. Ottenstein, "The program dependence graph in a software development environment," ACM Sigplan Notices, vol. 19, no. 5, pp. 177-184, 1984.

[18] S. Horwitz and T. Reps, "The use of program dependence graphs in software engineering," in Proceedings of the 14th international conference on Software engineering, 1992, pp. 392-411.

[19] A. Okutan and O. T. Yıldız, "Software defect prediction using Bayesian networks," Empirical Software Engineering, vol. 19, pp. 154-181, 2014.

[20] F. Subhan, X. Wu, L. Bo, X. Sun, and M. Rahman, "A deep learning-based approach for software vulnerability detection using code metrics," IET software, vol. 16, no. 5, pp. 516-526, 2022.

[21] G. Esteves, E. Figueiredo, A. Veloso, M. Viggiato, and N. Ziviani, "Understanding machine learning software defect predictions," Automated Software Engineering, vol. 27, no. 3, pp. 369-392, 2020.

[22] T. Siddiqui and M. Mustaqeem, "Performance evaluation of software defect prediction with NASA dataset using machine learning techniques," International Journal of Information Technology, vol. 15, no. 8, pp. 4131-4139, 2023.

[23] M. E. Aydin, « Leveraging Deep Learning for Enhanced Software Fault Prediction Using Error-type Metrics. »

[24] H. Liang, Y. Yu, L. Jiang, and Z. Xie, "Seml: A semantic LSTM model for software defect prediction," IEEe Access, vol. 7, pp. 83812-83824, 2019.

[25] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," Advances in neural information processing systems, vol. 32, 2019.

[26] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in Proceedings of the AAAI conference on artificial intelligence, 2016, vol. 30, no. 1.

[27] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," arXiv preprint arXiv:1711.00740, 2017.

[28] C. Pan, M. Lu, and B. Xu, "An empirical study on software defect prediction using codebert model," Applied Sciences, vol. 11, no. 11, p. 4793, 2021.

[29] H. Touvron et al., « Llama: Open and efficient foundation language models, » arXiv preprint arXiv:2302.13971, 2023.

[30] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers), 2019, pp. 4171-4186.

[31] M. Zaheer et al., "Big bird: Transformers for longer sequences," Advances in neural information processing systems, vol. 33, pp. 17283-17297, 2020.

[32] Z. Feng et al., « Codebert: A pre-trained model for programming and natural languages, » arXiv preprint arXiv:2002.08155, 2020.

[33] R. Li et al., « Starcoder: may the source be with you!, » arXiv preprint arXiv:2305.06161, 2023.

[34] T. Dao, D. Fu, S. Ermon, A. Rudra, and C. Ré, "Flashattention: Fast and memory-efficient exact attention with io-awareness," Advances in neural information processing systems, vol. 35, pp. 16344-16359, 2022.

[35] N. Shazeer, « Fast transformer decoding : One write-head is all you need, » arXiv preprint arXiv:1911.02150, 2019.

[36] D. Guo et al., « DeepSeek-Coder: When the Large Language Model Meets Programming--The Rise of Code Intelligence, » arXiv preprint arXiv:2401.14196, 2024.

[37] J. Su, M. Ahmed, Y. Lu, S. Pan, W. Bo, and Y. Liu, "Roformer: Enhanced transformer with rotary position embedding," Neurocomputing, vol. 568, p. 127063, 2024.

[38] M. Bavarian et al., « Efficient training of language models to fill in the middle, » arXiv preprint arXiv:2207.14255, 2022.

[39] J. Ainslie, J. Lee-Thorp, M. De Jong, Y. Zemlyanskiy, F. Lebrón, and S. Sanghai, "Gqa: Training generalized multi-query transformer models from multi-head checkpoints," arXiv preprint arXiv:2305.13245, 2023.