MONASH University

Context-Aware AI-Powered Code Review Assistant

Chanathip Pornprasit

Ph.D in Information Technology

A thesis submitted for the degree of Doctor of Philosophy at Monash University in 2025

Faculty of Information Technology

Copyright notice

Abstract

Software defects can cause trouble in real-world scenarios. Thus, software development teams adopt software quality assurance (SQA) to ensure that there are no software defects found in software systems after they are released. Modern code review is one of the SQA activities, where software developers review patches created by another software developer to ensure that software has high quality. Recent work has proposed various AI-powered code review assistants to help developers prioritize patches to be reviewed, and automate code review tasks. Unfortunately, contextual information has been largely ignored by many AI-powered code review assistants for code review. However, performing code review largely involves both technical information (i.e., the code that reviewers need to review) and contextual information (e.g., similar code, code characteristics, etc). Therefore, such AI-powered code review assistants have two limitations: (1) defect prediction models lack contextual information to generate the explanation of their predictions, and (2) approaches for code review automation lack contextual information to recommend the improved version of the submitted patch.

In this thesis, we hypothesize that AI-powered code review assistants should not solely rely on technical information, but also the contextual information of its technical information. To validate this hypothesis, we design, propose, develop, and evaluate context-aware AI-powered code review assistants. To do so, we consider and integrate various types of contextual information into our context-aware AI-powered code review assistants including (1) incorporating similar code characteristics to generate the explanation of the predictions generated by JIT defect prediction models, (2) incorporating code tokens from surrounding lines in a source code file to better localize the defective lines, and (3) incorporating the previous version of a patch and code from other patches similar to the current version of a patch to better recommend potential code improvement of the patch. The experimental results confirm the hypothesis that AI-powered code review assistants should not solely rely on technical information, but also on the contextual information of its technical information.

Declaration

This thesis is an original work of my research and contains no material which has been accepted for the award of any other degree or diploma at any university or equivalent institution and that, to the best of my knowledge and belief, this thesis contains no material previously published or written by another person, except where due reference is made in the text of the thesis.

Signature: ……

Print Name: Chanathip Pornprasit.

Date: 02/01/2025.

Publications during enrolment

- **Chanathip Pornprasit**, Chakkrit Tantithamthavorn, Jirayus Jiarpakdee, Michael Fu, and Patanamon Thongtanunam. “PyExplainer: Explaining the Predictions of Just-In-Time Defect Models.” In Proceedings of ASE. 2021, pp. 407–418.

- **Chanathip Pornprasit**, and Chakkrit Tantithamthavorn. “JITLine: A Simpler, Better, Faster, Finer-grained Just-In-Time Defect Prediction”. In Proceedings of MSR. 2021, pp. 369–379.

- **Chanathip Pornprasit**, and Chakkrit Tantithamthavorn. “Deeplinedp: Towards a deep learning approach for line-level defect prediction.” In IEEE Transactions on Software Engineering (TSE) 49.1 (2022), pp. 84–98.

- **Chanathip Pornprasit**, Chakkrit Tantithamthavorn, Patanamon Thongtanunam and Chunyang Chen.“D-ACT: Towards Diff-Aware Code Transformation for Code Review Under a Time-Wise Evaluation.” In Proceedings of SANER. 2023, pp. 296–307.

- **Chanathip Pornprasit**, and Chakkrit Tantithamthavorn. “Fine-tuning and prompt engineering for large language models-based code review automation.” Information and Software Technology (IST), 175, p.107523, 2024.

Thesis including published works declaration

I hereby declare that this thesis contains no material which has been accepted for the award of any other degree or diploma at any university or equivalent institution and that, to the best of my knowledge and belief, this thesis contains no material previously published or written by another person, except where due reference is made in the text of the thesis.

This thesis includes original papers published in peer reviewed conferences and journals. The core theme of the thesis is *Context-Aware AI-Powered Code Review Assistant*. The ideas, development and writing up of all the papers in the thesis were the principal responsibility of myself, the student, working within the Doctor of Philosophy (0190) under the supervision of Dr. Chakkrit Tantithamthavorn (Main supervisor).

The inclusion of co-authors reflects the fact that the work came from active collaboration between researchers and acknowledges input into team-based research.

In the case of Chapter 2, 3, 4, 5 and 6, my contribution to the work involved the following:

I have renumbered sections of submitted or published papers in order to generate a consistent presentation within the thesis.

Student name: Chanathip Pornprasit

Student signature: **Date:** 02/01/2025

I hereby certify that the above declaration correctly reflects the nature and extent of the student's and co-authors' contributions to this work. In instances where I am not the responsible author I have consulted with the responsible author to agree on the respective contributions of the authors.

Main Supervisor name: Chakkrit Tantithamthavorn

Main Supervisor signature: **Date:** 02/01/2025

Acknowledgements

I am very grateful to my supervisor, Dr. Chakkrit Tantithamthavorn, who has always guided me with great advice and support during my Ph.D. candidature. He expertly mentored me through producing high-quality research papers and making impressive presentations, to conduct myself professionally, and to live life wisely. The lessons learned from his suggestions are invaluable and will last long after leaving here as his Ph.D. student.

I am also indebted to my co-supervisor, Dr. Chunyang Chen, for the very perceptive comments and advice. Indeed, his time and efforts in our meetings have given substantive support to the further development and refinement of my work.

Many thanks to the chair, Prof. John Grundy, and panel members Dr. Li Li, Dr. Xiaoning Du, and Dr. Chetan Arora, for my milestone presentation; their constructive feedback and valuable suggestions played a very important role in improving the quality of my thesis.

Special acknowledgments go to Dr. Patanamon Thongtanunam from The University of Melbourne, and Assoc. Prof. Shane McIntosh at the University of Waterloo for their invaluable guidance. Dr. Patanamon on professional conduct and quality writing, along with Assoc. Prof. Shane on the art of delivering engaging and memorable presentations, has contributed enormously to my journey in academics.

I am particularly grateful to Dr. Jirat Pasuksmit and Dr. Jirayus Jiarpakdee for their technical insights and guidance during my research. Their general advice on life after graduation has been no less enriching, and I am quite confident that their tips will be useful in the future.

Last but not least, I would like to pass my big word of thanks to my family and friends for the support they have been giving me through my work in a Ph.D. study. I believe I could not successfully graduate without their support, which helped me to have more encouragement to overcome challenges that I encountered while working on my research projects.

Thanks to Julie Holden and FIT-GR staff for their academic support. Their help allowed me to be able to finish my Ph.D. successfully.

I am grateful for the financial support received from Monash University, through

the Monash Graduate Scholarship (MGS) and Monash International Tuition Scholarships (MITS), that has contributed towards financing this study (24/Jun/2021 - 22/Dec/2024).

Finally, I would like to acknowledge generative AI (i.e., ChatGPT), which helped me refine the Introduction and Conclusion sections of the thesis.

Contents

List of Tables

List of Figures

1 Introduction

Software defects are conditions in software systems that do not satisfy users' requirements or expectations. Software defects can occur for different reasons such as coding mistakes, inadequate software testing, or bad code review practices. In the past, several software defects that caused trouble in real-world scenarios were reported. For example, in 2023, software defects were found in the stock trading system of New York[1]. Due to these software defects, the system was halted since the stock price for an opening auction was miscalculated. Another example is the software defects in the banking system of a bank in Ireland, which was also found in 2023[2]. These software defects allowed customers to transfer and withdraw their money beyond their limits, and made online banking unavailable. From the above example, it is evident that software defects can cause trouble in real-world scenarios.

To ensure that there is no software defect found in software systems after they are released, software quality assurance (SQA) is widely adopted in software development teams. Software quality assurance is a process that ensures all processes, methods, activities and work items during software development comply with the standards defined by an organization. SQA includes various activities such as SQA planning, software testing (testing all source code before the software is released), and code review (reviewing submitted code changes before they are integrated into a codebase).

Modern code review is one of the practices in SQA to ensure high code quality. In modern code review, a patch author (i.e., a developer that creates a patch) first uploads a patch (i.e., a set of changes in source code) to a code review platform (e.g., Gerrit). Then, reviewers (i.e., developers other than patch authors) review patches (i.e., sets of modified, deleted, or newly created files) created by patch authors to provide feedback so that the created patches are improved until they meet quality standards and are approved to be integrated into codebase. Currently, modern code review practice is widely adopted in software industries (e.g., Microsoft [1] and Google [2]) and open-source software projects (e.g., Openstack [3] and Qt [4]).

[1] https://www.forbes.com/sites/jonathanponciano/2023/01/25/new-york-stock-exchanges-manual-error-briefly-wiped-billions-of-dollars-in-market-value-heres-what-happened/

[2] https://www.theguardian.com/business/2023/aug/16/bank-of-ireland-apologises-after-it-glitch-let-customers-withdraw-money-they-didnt-have

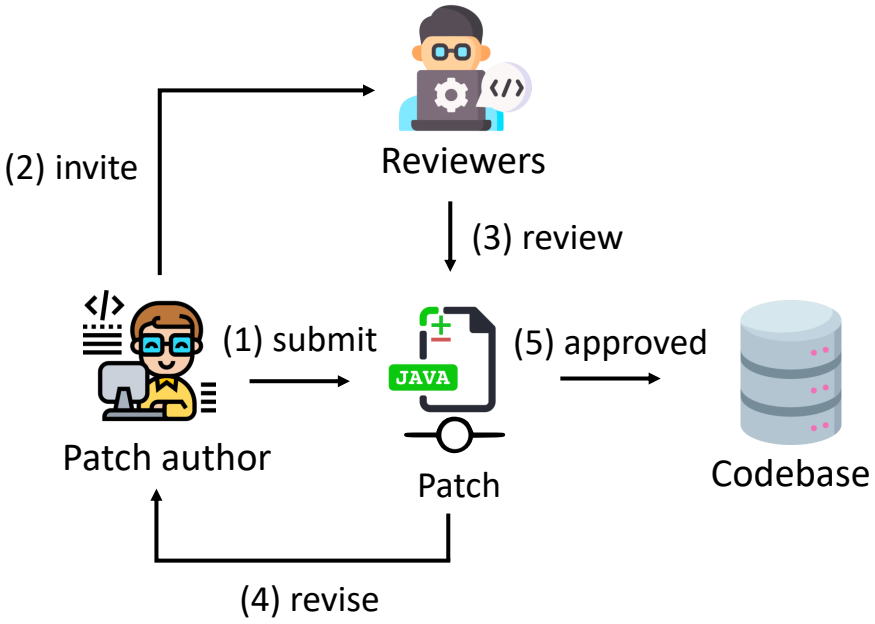


Figure 1: An overview of the code review process.

Figure 1 illustrates the modern code review process which consists of the following five main steps:

1. A patch author submits the first version of his/her patch to a code review platform (e.g., Gerrit).
2. The patch author invites reviewers to review and provide feedback on the submitted patch.
3. Reviewers review and provide feedback to the patch author
4. The patch author then revises the patch based on the reviewers' feedback, and submits the revised patch to the code review platform again.
5. If the reviewers agree that the revised patch has sufficient quality, they will approve the revised patch (i.e., the *approved version*) to be integrated into the codebase. Otherwise, step 3. - step 5. are repeated.

In large-scale software projects, developers have to undergo a lot of code review (e.g., developers in Microsoft Bing have to undergo around 3k code reviews per month [5]). In addition, developers spend a lot of time to perform code review. For example, developers in open-source software projects have to spend more than six hours per week on average reviewing code [6]. However, developer teams

have limited SQA resource (e.g., time). Thus, performing exhaustive code review activities for all code changes is infeasible.

To address the above practitioners' challenges, prior work has proposed a number of AI-powered code review assistants. An AI-powered code review assistant is a tool or an approach that leverages artificial intelligence to facilitate practitioners in the code review process. For example, prior studies proposed approaches to help reviewers effective prioritize SQA resources on the patches that are likely to introduce defects in future [7–12], or recommend the improved version of the submitted patch [13–17].

1.1 Thesis Motivation

Generally, when reviewers review a patch, the reviewers may need contextual information (i.e., additional textual or numeric information for a particular source code-related entity) to help them better understand the patch that they are reviewing [18–21]. Reviewers can obtain contextual information from various sources. For example, reviewers can read parts of the codebase relevant to the code in the patch that they are reviewing to understand the code in the submitted patch [18]. In addition, reviewers can find a high-level explanation of the patch to understand the rationale of the submitted patch [19]. Static analysis tools can also provide additional information relating to code quality to help reviewers review the submitted patch [20, 21]. The above examples highlight the importance of contextual information for reviewers when reviewing submitted patches.

While contextual information is important for reviewers, recent AI-powered code review assistants still ignore contextual information. In other words, such AI-powered code review assistants only take technical information (i.e., the code that reviewers need to review) into account. Due to a lack of contextual information, these assistants still have the following limitations. First, these assistants cannot explain the characteristics of the patches that are predicted as defective. Second, these assistants cannot locate defective lines of the patches or files that are predicted as defective. Third, these assistants may recommend code improvement for the lines that should remain unchanged [14]. Lastly, these assistants may not preserve coding convention or standards when recommending code improvement.

1.2 Thesis Hypothesis and Research Questions

The above four limitations of AI-powered code review assistants lead to the following hypothesis:

> *AI-powered code review assistants should not solely rely on the technical information, but also on contextual information of its technical information.*

To validate the hypothesis, we formulate the overarching research question (please note that at each stage of the project, we then pose a series of sub-questions that both motivate and examine the work of that chapter more closely):

> *How can we use contextual information to enhance the capabilities of AI-powered code review assistants?*

To address the overarching research question, we develop a context-aware AI-powered code review assistant. In particular, we integrate the following contextual information into our context-aware AI-powered code review assistant. First, we integrate the patches similar to the patch predicted as defective to enable our assistant to explain why a patch is predicted as defective, and which lines in a patch that are likely to introduce defects in future. Second, we integrate the source code from surrounding lines to explain which lines in a file that are likely to introduce defects in future. Third, we integrate the first version of the submitted patch to enable our assistant to recommend code improvement in proper lines. Fourth, we integrate source code similar to the one in the submitted patch to enable our assistant to preserve coding convention or standards when recommending code improvement.

1.3 Contributions to knowledge

The contributions relating integrating contextual information into our AI-powered code review assistant are as follows:

Contribution 1: We present PyExplainer, a local rule-based model-agnostic technique for explaining the predictions of machine learning-based JIT defect prediction models. We evaluate the performance of our PyExplainer approach and the existing model-agnostic technique for explaining the prediction of machine learning models called LIME [22]. The results of the study imply that the explanations generated by PyExplainer could help practitioners to understand the aspects that are associated with the risk of being defect-introducing commit.

This work has been published in the following paper:

Chanathip Pornprasit, Chakkrit Tantithamthavorn, Jirayus Jiarpakdee, Michael Fu, and Patanamon Thongtanunam. "PyExplainer: Explaining the Predictions of Just-In-Time Defect Models." In Proceedings of ASE. 2021, pp. 407–418.

Contribution 2: We present JITLine, a line-level defect prediction model to identify defective lines in a defective patch. We evaluate the performance of our JITLine with the existing deep learning-based JIT defect prediction approaches [11, 12] and the n-gram-based line-level defect prediction approach [23]. The results of the study imply that our JITLine approach may help practitioners to better prioritize defect-introducing commits and better identify defective lines. This work has been published in the following paper:

Chanathip Pornprasit, and Chakkrit Tantithamthavorn. "JITLine: A Simpler, Better, Faster, Finer-grained Just-In-Time Defect Prediction". In Proceedings of MSR. 2021, pp. 369–379.

Contribution 3: We present DeepLineDP, a line-level defect prediction model to identify defective lines in a defective file. We evaluate our DeepLineDP against other deep learning-based file-level defect prediction models. The results of the study imply that the attention of surrounding tokens in a code line can be used to locate defective lines. This work has been published in the following paper:

Chanathip Pornprasit, and Chakkrit Tantithamthavorn. "Deeplinedp: Towards a deep learning approach for line-level defect prediction." In IEEE Transactions on Software Engineering (TSE) 49.1 (2022), pp. 84–98.

Contribution 4: We present D-ACT, a Neural Machine Translation-based code transformation approach to transform the first version of a patch to the version that is reviewed and approved by reviewers. We evaluate our D-ACT with the existing code transformation approaches [14, 24]. The results of the study imply that the token-level code difference information in our D-ACT approach can substantially improve the performance of NMT-based code transformation approaches for code review. This work has been published in the following paper:

Chanathip Pornprasit, Chakkrit Tantithamthavorn, Patanamon Thongtanunam and Chunyang Chen."D-ACT: Towards Diff-Aware Code Transformation for Code Review Under a Time-Wise Evaluation." In Proceedings of SANER. 2023, pp.

296–307.

Contribution 5: We conduct an experimental study to investigate the performance of LLMs-based code review automation when LLMs are leveraged by the following prompting techniques: zero-shot learning, few-shot learning and persona. The results of the study imply that few-shot learning without a persona should be used for LLMs for code review automation. The experimental results are published in the following paper:

Chanathip Pornprasit, and Chakkrit Tantithamthavorn. "Fine-tuning and prompt engineering for large language models-based code review automation." Information and Software Technology (IST), 175, p.107523, 2024.

The results of the above studies show that such contextual information being considered from various sources can help AI-powered code review assistants in generating the explanation of the prediction, generated by defect prediction models, specific to a file/patch; and substantially improve the performance of AI-based approaches for code review.

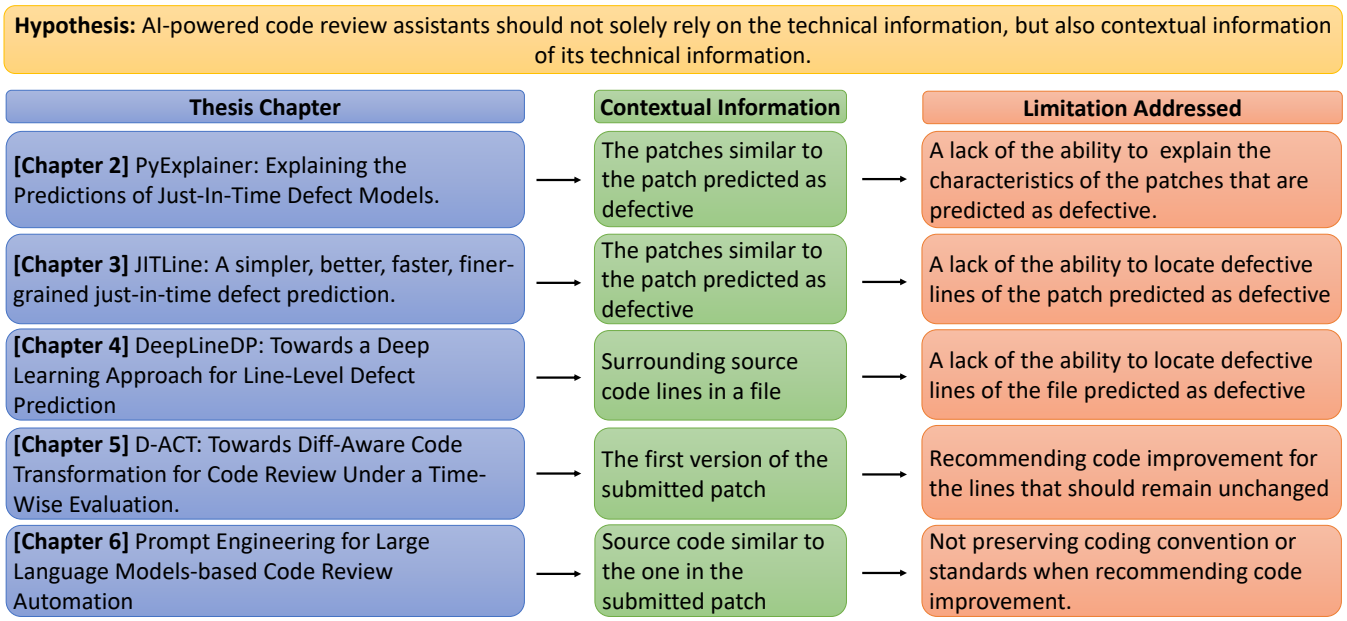


Figure 2: An overview of the thesis.

1.4 Thesis overview

In this subsection, we provide an overview of each chapter and its contributions to the thesis. Figure 2 outlines the overview of the thesis. The overview of each chapter is as follows.

[Chapter 2] PyExplainer: Explaining the Predictions of Just-In-Time Defect Models.

In Chapter 2, we present PyExplainer—i.e., a local rule-based model-agnostic technique for generating explanations of JIT defect models. We propose PyExplainer approach to address the following limitations of LIME (a state-of-the-art model-agnostic technique [22]): (1) the LIME's neighborhood generation process is still suboptimal, (2) the approximation of the LIME local models to the predictions of the global model is still suboptimal, and (3) the explanations generated by LIME are still not specific to an instance to be explained. The experimental results imply that our PyExplainer is more effective in generating explanations than LIME for the predictions of JIT defect prediction models.

[Chapter 3] JITLine: A simpler, better, faster, finer-grained just-in-time defect prediction.

In Chapter 3, we present JITLine—a Just-In-Time defect prediction approach for predicting defect-introducing commits and identifying lines that are associated with that defect-introducing commit (i.e., defective lines). We evaluate the JITLine approach with the state-of-the-art commit-level JIT defect prediction approaches (i.e., EARL [25], DeepJIT [11], and CC2Vec [12]) with respect to six traditional measures (i.e, AUC, F-measure, False Alarm Rate, Distance-to-Heaven, Precision, and Recall), three cost-effectiveness measures (i.e., PCI@20%LOC, Effort@20%Recall, P_{Opt}). In addition, we compare the JITLine approach with a baseline line-level JIT defect localization by Yan [23] using four line-level effort-aware measures (i.e., Top-10 Accuracy, Recall@20%LOC, Effort@20%$Recall_{line}$, Initial False Alarm). The results show that the JITLine approach is better, more cost-effective, faster and more fine-grained than the existing JIT defect prediction approaches (i.e., EARL, DeepJIT, and CC2Vec). In addition, based on our experimental results, we discuss the implications for practitioners and researchers.

[Chapter 4] DeepLineDP: Towards a Deep Learning Approach for Line-Level Defect Prediction

In Chapter 4, we present the DeepLineDP approach for predicting defect-introducing files and identifying lines that are associated with that defect-introducing file. We propose DeepLineDP to address the following limitations in previous file-level defect prediction approaches: (1) the granularity levels of defect predictions are still coarse-grained and (2) the surrounding tokens and surrounding lines have

not yet been fully utilized. We conduct an empirical study to demonstrate that our DeepLineDP approach is more accurate than other file-level defect prediction approaches and more cost-effective than other line-level defect prediction approaches.

[Chapter 5] D-ACT: Towards Diff-Aware Code Transformation for Code Review Under a Time-Wise Evaluation.

In Chapter 5, we present a Diff-Aware Code Transformation (D-ACT) approach for code review that leverages the token-level code difference information and CodeT5 [26] (the existing pre-trained code model). The D-ACT approach is proposed to address the following limitations in prior work: (1) a lack of Code-Diff Information, and (2) a lack of Temporal Information. We also collect a new dataset for our D-ACT approach. Our experimental results show that our D-ACT approach outperforms the existing code review automation approaches [14, 15].

[Chapter 6] Prompt Engineering for Large Language Models-based Code Review Automation

In Chapter 6, we conduct an experimental study to investigate the performance of LLMs-based code review automation when LLMs are leveraged by the following prompting techniques: zero-shot learning, few-shot learning and persona. In few-shot learning, we incorporate code from other patches that are similar to the current version of the submitted patch into the LLMs. The experimental results show that LLMs that are prompted by few-shot learning without a persona achieve the highest performance when compared to other prompting techniques in the study.

2 PyExplainer: Explaining the Predictions of Just-In-Time Defect Models

Chapter Overview

Just-In-Time (JIT) defect prediction approaches have been proposed to help developers prioritize their limited SQA resources during the code review process. However, the predictions of existing JIT defect prediction approaches are still not explainable. To address this problem, LIME – a model-agnostic technique can be used to explain the predictions of JIT defect prediction approaches. Nevertheless, prior work found that LIME may not be suitable for explaining the predictions of models trained JIT defect datasets.

In this chapter, to address the overarching research question of the thesis, we propose PyExplainer that incorporates the characteristics of the patches similar to the patch that JIT defect prediction models predict as defective to generate a rule-based explanation of the predictions. Through a case study of two open-source software projects, we find that our PyExplainer produces more similar synthetic neighbors, more accurate local models, and more unique explanations than LIME.

The work in this chapter appears in Chanathip Pornprasit, Chakkrit Tantithamthavorn, Jirayus Jiarpakdee, Michael Fu, and Patanamon Thongtanunam. “PyExplainer: Explaining the Predictions of Just-In-Time Defect Models.” In Proceedings of ASE. 2021, pp. 407–418.

2.1 Introduction

Today software projects are globally-distributed large-scale. In addition, modern software development projects tend to release software products in rapid cycles. To ensure the quality of all newly arrived commits, developers need to conduct code review and provide feedback prior to merging them into the release branch. However, such code review activities are still time-consuming and expensive. Thus, performing exhaustive code review activities for all commits is infeasible due to the limited Software Quality Assurance (SQA) resources.

Just-In-Time (JIT) defect prediction [25, 27–29]—an AI/ML model to predict defect-introducing commits—has been proposed to help developers efficiently prioritize their limited SQA resources on the most risky commits. In addition, JIT defect prediction is also used to provide insights about the important characteristics of defect-introducing commits. Such insights can help QA teams and managers to develop proactive software quality improvement plans to prevent pitfalls in the past that lead to software defects in the future [30].

However, the predictions of existing JIT defect prediction approaches are still not explainable, hindering the adoption of JIT defect models in practice [31–34]. Recent research shows that practitioners still asked many *why*-questions (e.g., why a commit is predicted as defective) [31–33, 35], since current JIT defect prediction approaches are treated as black-box which only provide the predictions, not the explanations. Such a lack of explainability of JIT defect prediction approaches could lead to suboptimal software quality assurance practices and suboptimal operational decision-makings.

Recently, LIME—a state-of-the-art model-agnostic technique [22]—has been adopted in software engineering research (e.g., line-level just-in-time defect prediction [28], and explainable file-level defect prediction [32]). LIME is a technique that explains a prediction of AI/ML models (i.e., what are the features that influence a given prediction). Generally speaking, given an instance to be explained (e.g., a commit), LIME produces an explanation from a local model (F') that is trained using the randomly generated synthetic instances around the instance to be explained (X') (i.e., *synthetic neighbors*) and the predictions (Y') obtained from the global black-box model. This allows the local model to mimic the behavior of the underlying global black-box models.

The quality of explanation produced by LIME heavily relies on the neighborhood

generation process [36]. Ideally, the neighborhood generation process should generate synthetic neighbors that are closely similar to the instance to be explained so that the local model can accurately approximate the prediction of the global models. In LIME, the random perturbation approach is used to generate synthetic neighbors. However, such a simple random perturbation approach may not be suitable for sparse and high dimensional data like JIT datasets [37]. It is possible that the random perturbation approach will generate synthetic neighbors that may not be similar to an instance to be explained, which will lead the local model to inaccurately approximate the predictions of the global model. Thus, these local models may not be effective in generating explanations (e.g., generic).

In this chapter, we propose PyExplainer, a local rule-based model-agnostic technique for explaining the predictions of JIT defect prediction models. To produce a more accurate explanation for the prediction of JIT defect models, our PyExplainer generates synthetic neighbors based on the actual characteristics of defect-introducing commits in the JIT dataset using the crossover and mutation operations. Instead of generating an explanation with a single rule feature with an importance score like LIME (e.g., the importance score of Churn$>$100 is 0.9), our PyExplainer generates an explanation that accounts for interactions between rule features (e.g., Churn$>$100 & #Reviewers$<$2 $\Rightarrow$ Defect).

To evaluate our PyExplainer, we compare with LIME [22] in three dimensions: (1) *the similarity between synthetic neighbors and an instance to be explained*; (2) *the accuracy of the local models*; and (3) *the effectiveness in generating explanations*. Through a case study of 40,798 commits that span across two large-scale software systems (i.e., OpenStack and Qt), we address the following three research questions:

(RQ1) Does our PyExplainer produce better synthetic neighbors than LIME for JIT defect models?
The synthetic neighours produced by our PyExplainer are 41%-55% more similar to an instance to be explained than LIME, indicating that our PyExplainer produces synthetic neighbors that are more closely similar to an instance to be explained than LIME.

(RQ2) Does our PyExplainer produce higher accuracy of local models than LIME for JIT defect models?
When explaining the RF and LR JIT defect models, PyExplainer produces

local models that are 18%-38% more accurate (AUC) than the local models produced by LIME, indicating that the PyExplainer produces local models that have a higher ability to discriminate the characteristics between defect and clean classes.

(RQ3) Is our PyExplainer more effective in generating explanations than LIME for JIT defect models?
The explanations generated by our PyExplainer are 69%-98% more unique (i.e., more specific to an instance to be explained) than LIME. On the other hand, the explanations generated by our PyExplainer are 17%-54% more consistent with the actual defect-introducing commits in the testing data than LIME.

Thus, the explanations generated by PyExplainer could help practitioners to focus on the most important aspects that are associated with the risk of being defect-introducing for a given prediction, instead of focusing on the less important aspects.

Contributions. The contributions of our work in this chapter are as follows:

- We propose PyExplainer, a local rule-based model-agnostic technique for explaining the predictions of JIT defect models.
- Our results show that PyExplainer produces (1) synthetic neighbours that are more similar to an instance to be explained; (2) more accurate local models; and (3) explanations that are more unique and more consistent with the actual characteristics of defect-introducing commits in the future than LIME.
- Finally, we developed a proof-of-concept of visual explanations and *what-if* visualizations, and published our PyExplainer as a python package.

2.2 Related Work & Research Questions

Prior studies pointed out that practitioners often do not understand the reasons behind the predictions of software analytics [31, 33, 38]. Recent work also raises concerns that a lack of explainability of software analytics often hinder the adoption of software analytics in practice [31–35, 39]. Importantly, Jiarpakdee *et al.* [33] found that 91% of recent defect prediction studies often focus on improving the accuracy, while as few as 4% of recent defect prediction studies

focus on making file-level defect prediction models more explainable. However, Jiarpakdee *et al.* [33] found that practitioners perceived that providing the explanations of defect prediction models are as important and useful as improving the accuracy of defect prediction models. Yet, the explainability of JIT defect models remains largely unexplored.

The explanability of software analytics can be achieved at the global and the local levels.

The *global* explanation can be produced using model-specific interpretation techniques that are built in the AI/ML models (e.g., ANOVA for regression analysis, variable importance analysis for random forest). This explanation helps researchers and software practitioners understand what important features that influence the predictions of the models [40–44]. However, this global explanation is not specific to the prediction of each instance (e.g., a commit) in the testing or unseen data, since the global explanation is derived from the training dataset [45]. Hence, the global explanation may not be accurate for a particular prediction.

On the other hand, the *local* explanation is produced for a particular prediction of an instance in the testing or unseen dataset, which allow practitioners better understand the reasons behind the predictions of the AI/ML models [32]. **LIME** [22] is a state-of-the-art model-agnostic technique which has been widely adopted to address various software engineering problems and other domains (5,000+ citations). For example, recent work [28, 45] employed LIME for line-level defect predictions (i.e., identifying defective lines that contain the risky code tokens explained by LIME). Jiarpakdee *et al.* [32] found that LIME [22] is effective in explaining the predictions of file-level defect prediction models (i.e., why a file is predicted as defective). However, LIME has the following limitations.

First, the LIME's neighborhood generation process is still suboptimal. The quality of an explanation for a prediction heavily relies on the quality of the generated synthetic neighbors around the instance to be explained [46]. Thus, if the neighbor generation process is suboptimal, the local model may fail to provide accurate insights about the logical reasoning of the global model. Jia *et al.* [36] found that the size of the neighbourhood has a large impact on the quality of the explanation. Krishnan *et al.* [47] found that when a model is learned from sparse and high dimensional data (e.g., just-in-time defect dataset [37]), the model is often underfitting, failing to capture the phenomenon of the data being trained. Thus, the neighbor generation process should ideally generate synthetic neigh-

bors that are similar to the instance to be explained. Therefore, we investigate whether our PyExplainer produce better synthetic neighbors (i.e., the synthetic neighbors that are more similar to an instance to be explained) than LIME for JIT defect models. We formulate the following RQ:

(RQ1) Does our PyExplainer produce better synthetic neighbors than LIME for JIT defect models?

Second, the approximation of the LIME local models to the predictions of the global model is still suboptimal. One of the key principles of model-agnostic techniques is to build the best local model to mimic the behavior of the predictions of the global models. Accuracy is often used to measure the extent to which how well the local model can approximate the predictions of the global model [22]. Thus, a high accuracy of local models is desirable in order to derive the highest quality explanation, i.e., the local models can accurately approximate the global model predictions for a subset of the data (e.g., local surrogate models). Since LIME uses the random perturbation method to generate synthetic neighbors, the approximation of the LIME local models to the predictions of the global model may be suboptimal, leading to the inaccurate local models produced by LIME. Thus, we formulate the following RQ:

(RQ2) Does our PyExplainer produce higher accuracy of local models than LIME for JIT defect models?

Third, the explanations generated by LIME are still not specific to an instance to be explained. Another key principle of the model-agnostic techniques is to build a unique explanation that is specific to the prediction of an instance to be explained. Thus, explanations should be unique and highlight the key characteristics (i.e., features) of a commit that leads a global model to predict as defect-introducing. However, LIME uses a Quantile discretization (i.e., 1st, 2nd, 3rd Quantiles) for generating rule features. The three bins used by LIME may not be enough to capture the highly-complex and highly-skewed JIT defect datasets. Thus, the rule features used by LIME may produce generic explanations that may not be specific to the instance to be explained, which may not reflect the actual characteristics of defect-introducing commits. Therefore, we formulate the following RQ:

(RQ3) Is our PyExplainer more effective in generating explanations than LIME for JIT defect models?

2.3 Our PyExplainer: A Local Rule-Based Model-Agnostic Technique

In this section, we present our PyExplainer, a local rule-based model-agnostic approach for explaining the predictions of JIT defect models.

Overview. Figure 3 illustrates an overview of the PyExplainer approach, which consists of four main steps. First, given an instance to be explained (i.e., a commit) and a global model, our PyExplainer will generate synthetic neighbors around the instance to be explained using the crossover and mutation techniques [48]. Second, our PyExplainer will obtains the predictions of the synthetic neighbors from the global model. Third, our PyExplainer builds a local rule-based regression model in order to learn the associations between the characteristics of the synthetic instances and the predictions from the global model. In the fourth step, our PyExplainer generates an explanation from the local model for the instance to be explained. We now describe each of the four steps below.

(Step 1) Generate Synthetic Neighbors Around the Instance to be Explained. Our PyExplainer will first generate synthetic neighbors (X') around the instance to be explained using the crossover and mutation techniques [48]. To do so, PyExplainer will find an initial set of actual neighbors, i.e., the actual instances around the instance to be explained in the training dataset. To identify the actual neighbors, PyExplainer applies the exponential kernel function (see Eq. 1) to calculate the similarity score between each instance in the training dataset (i_k) and the instance to be explained (i_e).

$$K(i_k, i_e) = \text{exp}(-\frac{\text{dist}(i_k, i_e)^2}{2w^2}) \tag{1}$$

where dist(i_k, i_e) is the euclidean distance between instances i_k and i_e, and w is the kernel width as the multiplication of 0.75 and the number of features of an instance ($w = 0.75 \times$ #features) as suggested by Ribeiro *et al.* [22].

Based on the initial set of actual neighbor, PyExplainer generates synthetic neighbors using crossover and mutation techniques to expand the initial set. The calculation of the crossover ($I_{\text{crossover}}$) and mutation (I_{mutation}) techniques can be derived as follows:

$$I_{\text{crossover}} = i_1 + (i_2 - i_1) * \alpha \tag{2}$$

$$I_{\text{mutation}} = i_1 + (i_2 - i_3) * \mu \tag{3}$$

where i_1, i_2, and i_3 are the randomly selected neighbourhood instances, α is a randomly generated number between 0 and 1; and μ is a randomly generated

number between 0.5 and 1.

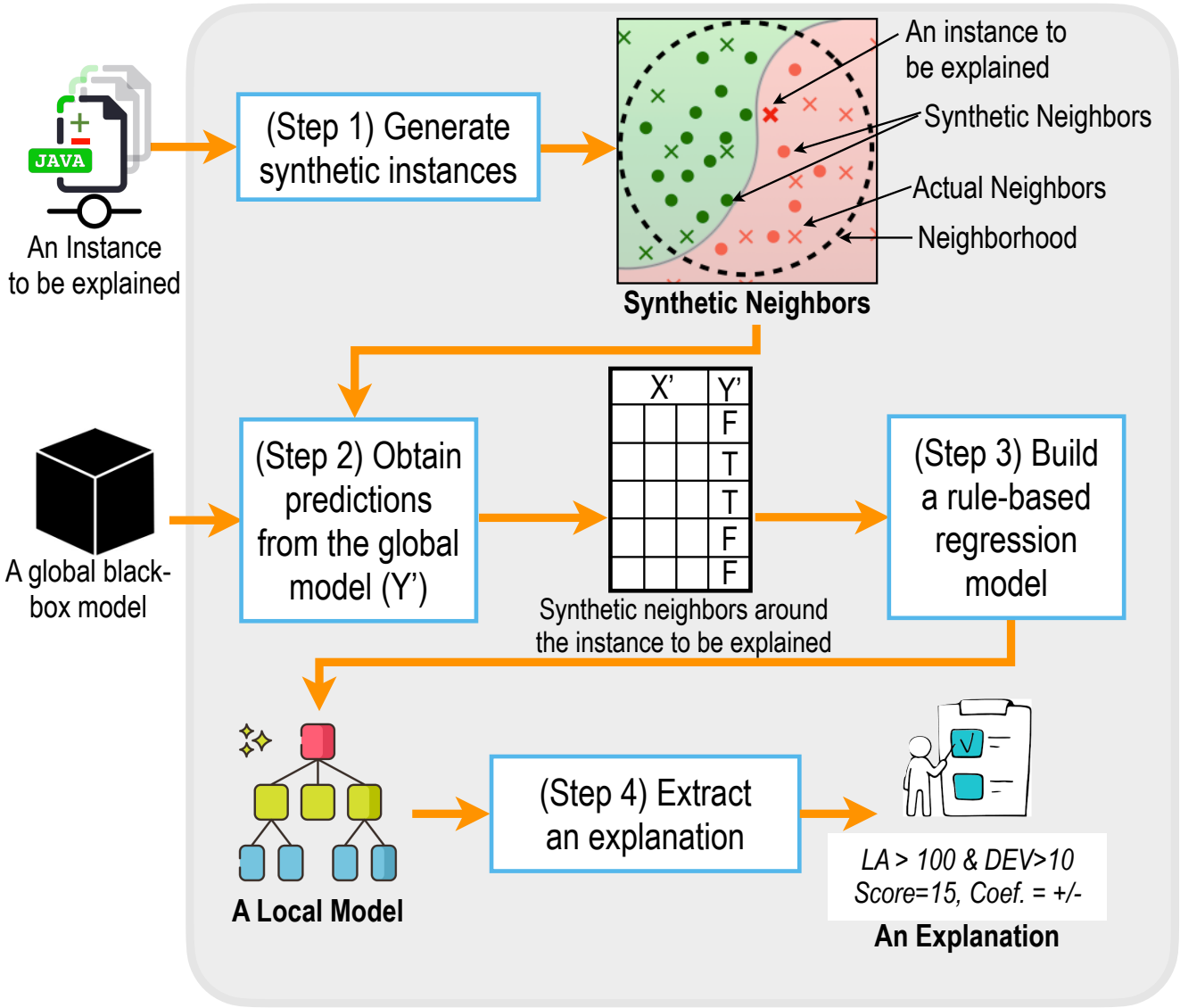


Figure 3: An overview of the PyExplainer approach. Given an instance to be explained, PyExplainer produces four main component i.e., (1) synthetic neighbors, (2) a local model, and (3) an explanation. Each PyExplainer's explanation produces three pieces of information i.e., (1) a rule-based explanation, (2) an importance score, and (3) the direction of relationship of either supporting (+) or contradicting (-) the prediction.

(Step 2) Obtain the Predictions of the Synthetic Instances using the Global Model. In Step 1, only the features of a synthetic neighbor (X') are generated. Hence, PyExplainer uses the global model to obtain the predictions (i.e., whether it is defective or clean given features of a synthetic instance). This allows PyExplainer to learn the behaviour of the underlying global model.

(Step 3) Build a Local Rule-based Regression Model using the RuleFit tech-

Table 1: An overview of the studied JIT defect datasets provided by McIntosh and Kamei [30].

	Training Data				Testing Data			
Project	**Start Date**	**End Date**	**# Commits**	**# Defective Commits**	**Start Date**	**End Date**	**# Commits**	**# Defective Commits**
Openstack	11/30/2011	08/13/2013	9,246	980 (11%)	08/13/2013	02/28/2014	3,963	646 (16%)
Qt	06/18/2011	05/08/2013	19,312	1,577 (8%)	05/08/2013	03/18/2014	8,277	476 (6%)

nique. To build a local model (F'), PyExplainer uses a rule-based logistic regression technique, called RuleFit [49]. RuleFit is a classifier that combines tree ensembles and linear models, which allows us to interpret the model like a traditional regression model, while understanding the logical reasons learnt from the rule features.

Broadly speaking, RuleFit will first generate *rule features* (X'_R), e.g., {Churn $>$ 100 & #Reviewers $<$ 2} based on ensemble decision trees (e.g., Gradient Boosting Trees). Then, RuleFit uses a regression model (i.e., logistic regression for binary outcomes, or linear regression for continuous outcomes) to model the association between the predicted outcomes (Y') and the rule features (X'_R) together with the original features (X). Then, the degree of importance and the coefficients of rule features and the original features can be analyzed from this regression model.

The use of RuleFit in our PyExplainer will address the limitation of LIME which does not account for interactions between features (i.e., the combination of rule features). Although existing rule-based model-agnostic techniques (e.g., SQAPlanner [50], Anchors [51], and LORE [52]) have been proposed, these techniques employed association rule mining techniques (e.g., Apriori, FP-Growth) which do not provide the degree of importance of the rules and the coefficients. Without the degree of importance of the rules and the coefficients provided by such techniques, we cannot quantify how strong the association between the rules and the predicted outcome and what the direction of the relationship is.

(Step 4) Extract an Explanation from the Local Rule-based Model. To generate an explanation, our PyExplainer analyzes the local model which is built using the RuleFit technique in Step 3. In the local model, there are three key pieces information: (1) rule features, (2) importance scores, and (3) coefficients. The importance score indicates the strength of the association between the rule feature and the predicted outcome. The coefficient can be used to indicate the direction of the relationship. For example, a positive coefficient indicates that a rule feature has a contribution towards the prediction of the TRUE class (i.e., DEFECT).

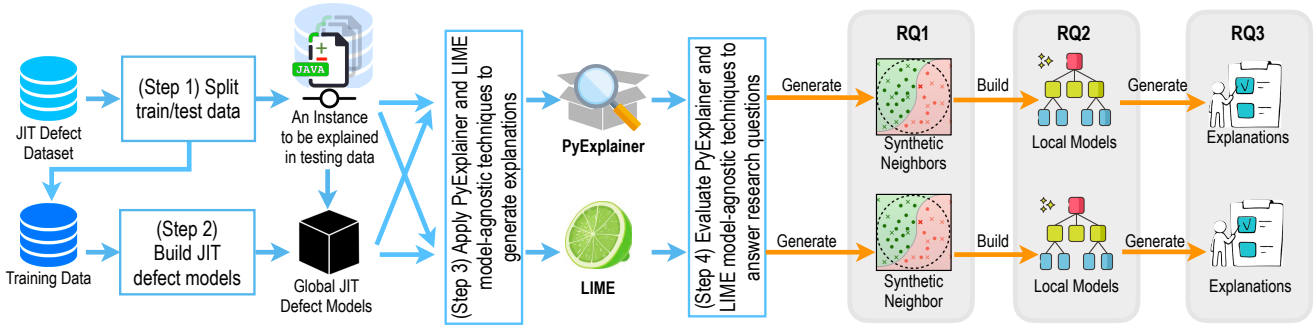


Figure 4: An overview of the experimental design.

Based on the three key pieces of information in the local model, our PyExplainer generates an explanation by identifying the rule feature that has the highest importance score, has a positive coefficient, and satisfies the actual feature values of the instance to be explained. For example, suppose that a rule feature (Churn$>$100 & #Reviewers$<$2) has the highest importance score and has a positive coefficient, our PyExplainer will generate the following rule-based explanation: Churn$>$100 & #Reviewers$<$2 $\Rightarrow$ DEFECT, which means that a commit is predicted as defective since Churn is greater than 100 and the number of reviewers is less than 2.

2.4 Experimental Design

In this section, we present the studied datasets and explain the details of our experimental design.

Studied JIT Datasets. We select just-in-time defect datasets from two large-scale open-source software projects (i.e., Openstack and Qt) as provided by McIntosh and Kamei [30]. Openstack is an open-source software for cloud infrastructure service. Qt is a cross-platform application development framework. We choose Openstack and Qt datasets for our study, since both datasets (1) are often used as a benchmark in defect prediction studies [11, 12, 28, 30]; and (2) are manually verified for the validity of the SZZ algorithm [53] to reduce the number of false positives and false negatives [54, 55]. Table 1 provides an overview of the studied datasets.

Commit Features. For each dataset, there are 17 commit-level features that span across 5 dimensions, i.e., Size (e.g., lines added, lines deleted), Diffusion (e.g., #modified files), History (#developers), Experience, and Code Review Activities.

Experiment Design. Figure 4 presents an overview of our experimental design, which is composed of four main steps.

(Step 1) Split Data into Training and Testing Datasets. To ensure that the evaluation of our just-in-time defect prediction reflects a real-world scenario, we first sort the date of the commits to preserve the order of the commits in a chronological order [28, 56]. Then, we use a time-wise hold-out validation technique (as used by McIntosh and Kamei [30]) to split the dataset into training (70%) and testing (30%) datasets. The use of the time-wise hold-out validation technique ensures that the commits that appear later will not be used in model training. Similarly, the commits that appear earlier will not be used in model evaluation.

(Step 2) Build JIT Defect Models. For each training dataset, we first mitigate collinearity using AutoSpearman and handle class imbalance using SMOTE prior to build JIT defect models. Below, we describe each step in detail.

(Step 2-1) Mitigate Collinearity using AutoSpearman. To ensure that the interpretation of our JIT defect models is highly accurate, we mitigate collinearity and multi-collinearity, as suggested by prior studies [57–59]. We use *AutoSpearman*, an automated feature selection approach to automatically select one feature from each group of highly correlated features that shares the least correlation with the other features that are not in the group [57]. As suggested by Kraemer *et al.* [60], we use a threshold of 0.7 to indicate strong correlation between features. As suggested by Fox [61], we use a VIF threshold value of 5 to indicate multicollinearity. We use the implementation of *AutoSpearman* as provided by the `PyExplainer` Python package. After using AutoSpearman, we finally select 7 features that are not highly-correlated with each other.

(Step 2-2) Handle Class Imbalance using SMOTE. To ensure that the predictions of our JIT defect models are highly accurate, we apply a class rebalancing technique, as suggested by prior work [62, 63]. Since the defective ratio of our studied JIT defect datasets are highly imbalanced (i.e., 8%-16%), we apply SMOTE [64] to handle class imbalance *only on the training dataset*. We choose the SMOTE technique, as suggested by prior work [62, 63] who found that the SMOTE technique outperforms other class rebalancing techniques. SMOTE performs the following steps. First, SMOTE calculates the k-nearest neighbors of a set of minority class. Then, SMOTE randomly chooses the neighbors and generates synthetic instances around such neighbors. Finally, SMOTE combines the synthetic instances with the undersampling of the majority class to produce the final

set of balanced instances. We use the implementation of SMOTE as provided by `Imbalanced-Learn` Python library [65]. We use the default setting ($k = 5$) of SMOTE, since our experiment with various k settings has shown that varying the k settings has little impact on the performance of our JIT defect models.

(Step 2-3) Evaluate Global JIT Defect Models. We build global JIT defect models using the training data of each project. We select the two classification techniques that are commonly-used in prior studies [9, 25], since they found that Random Forests (RF) and Logistic Regressions (LR) often outperform other classification techniques. Then, we evaluate the global JIT defect models on the testing dataset using 2 effort-aware measures (i.e., Recall@20%effort, and P_{opt}) and 2 traditional performance measures (i.e., Area Under the ROC Curve (AUC) and F1 (with a cutoff threshold of 0.5). We select classifiers using Recall@20%effort to ensure that they are practical when they are deployed in practices [66]. Recall@20%effort measures the percentage of correctly predicted defect-introducing commits that can be found when inspecting the top-20% LOC of the most risky commits.

Our global JIT defect models trained using both RF and LR techniques achieve similar performance for both OpenStack and Qt projects. Table 2 presents the accuracy of the JIT defectt models. For Openstack, our RF classifier achieves a Recall@20%Effort of 0.56 for RF and 0.54 for LR, indicating that our JIT defect models can correctly predicted 54%-56% of defect-introducing commits when spending only 20% code inspection effort (i.e., LOC). Similarly, for Qt, our RF classifier achieves a Recall@20%Effort of 0.83 for RF and 0.82 for LR, indicating that our JIT defect models can correctly predicted 82%-83% of defect-introducing commits when inspecting only 20% code inspection effort (i.e., LOC).

(Step 3) Apply our PyExplainer and the LIME model-agnostic techniques to generate explanations. For each prediction of JIT defect models, we apply our PyExplainer and LIME to generate an explanation of each prediction. We choose LIME as a baseline comparison, since LIME has been widely used in SE research [28, 32, 45, 67] Similar to PyExplainer, LIME produces three main components: (1) synthetic neighbors; (2) local models; and (3) explanations. LIME performs the following four steps to produce an explanation.

First, LIME randomly generates synthetic neighbors surrounding an instance to be explained using a random perturbation method with an exponential kernel function of euclidean distance. Second, LIME obtains the predictions of the

Table 2: The accuracy of JIT defect models that are trained using Random Forest (RF) and Logistic Regression (LR).

	OpenStack			
Classification	**Recall@20%Effort**	**Popt**	**AUC**	**F1**
Random Forest	0.56	0.82	0.75	0.36
Logistic Regression	0.54	0.82	0.66	0.36
	Qt			
Classification	**Recall@20%Effort**	**Popt**	**AUC**	**F1**
Random Forest	0.83	0.94	0.74	0.21
Logistic Regression	0.82	0.95	0.64	0.16

synthetic neighbors from the global JIT defect models. Third, LIME builds a local sparse linear regression model (K-Lasso) using the randomly generated instances and their predictions from the global JIT defect models. Forth, LIME generates an explanation using the coefficients of the local K-Lasso model with three key pieces of information: (1) a decision rule feature; (2) the importance score; and (3) the direction of relationship of either supporting (+) or contradicting (-) the prediction towards a TRUE class (i.e., Defect).

(Step 4) Evaluate the PyExplainer and LIME model-agnostic techniques. Both PyExplainer and LIME use different techniques to generate synthetic neighbors (i.e., crossover and mutation vs. random perturbation) and the local models (i.e., RuleFit vs. K-Lasso). Thus, they may produce different explanations. Therefore, we aim to investigate which model-agnostic technique is the best to generate an explanation of the prediction obtained from JIT defect models. To evaluate PyExplainer and LIME, we focus on the three main components along two dimensions. In the first dimension, we focus on the common internal mechanism of the model-agnostic techniques i.e., the synthetic neighbour and the accuracy of their local models, since these two components are used to generate an explanation for a prediction. In the second dimension, we focus on the explanations generated by PyExplainer and LIME. We describe the analysis approach for each research question in Section 2.5.

2.5 Experimental Results

In this section, we present the approach, and results with respect to our three research questions.

(RQ1) Does our PyExplainer produce better synthetic neighbors than LIME for JIT defect models?

Approach. To address RQ1, we analyze the distance between an instance to be explained and the synthetic instances around the neighbourhood using the Euclidean Distance measure. The Euclidean Distance measure is the calculation of distance between two feature vectors in an n-dimensional feature space (i.e., $d(i_1, i_2) = \sqrt{\sum_{j=1}^{n}(i_{1j} - i_{2j})^2}$, where $d(i_1, i_2)$ is a Euclidean Distance of two instances i_1 and i_2). The smaller the distance, the higher similarity of the both vectors (instances). Thus, the lower distances between an instance to be explained and the synthetic instances indicate that such generated synthetic instances yield high similarity with the instance to be explained.

For each instance to be explained in the testing dataset, we calculate the Euclidean Distance between the instance to be explained and their synthetic instances. Since the data is not normally distributed, we compute the median value (instead of the average) of the Euclidean Distance to approximate the average similarity of the instances around the neighbourhood. Then, we compare the distributions of the median Euclidean Distance of the synthetic neighbors produced by both PyExplainer and LIME.

Finally, we apply two statistical test (i.e., Wilcoxon singed-rank test and Cliff's $|\delta|$ effect size) to identify whether differences of the Euclidean Distance produced by PyExplainer and LIME are statistically significant. The Wilcoxon signed-rank test is a non-parametric test that measures the difference of distribution between two population (i.e., the Euclidean Distance of PyExplainer and LIME). Cliff's $|\delta|$ is a non-parametric effect size test that measures the magnitude of the differences of the given two distributions. We use the Cliff's $|\delta|$ interpretation of Romano *et al.* [68] as follows, i.e., negligible for $|\delta| \leq 0.147$, small for $|\delta| \leq 0.33$, medium for $|\delta| \leq 0.474$, and large for $|\delta| > 0.474$ Finally, we compute the relative percentage difference using the following equation: $\%\text{diff} = \frac{\text{PyExplainer} - \text{LIME}}{\text{LIME}} \times 100$.

Results. The synthetic neighours produced by our PyExplainer is 41%-55% more similar to an instance to be explained than LIME for both RF and LR JIT defect models. Figure 5 shows that our PyExplainer achieves 41%-49% and 47%-55% lower Euclidean Distance for both RF and LR JIT defect models for both Openstack and Qt. For Openstack, we find that our PyExplainer achieves a median Euclidean Distance of 492, while LIME achieves a median Euclidean Distance 839. For Qt, we find that our PyExplainer achieves a median

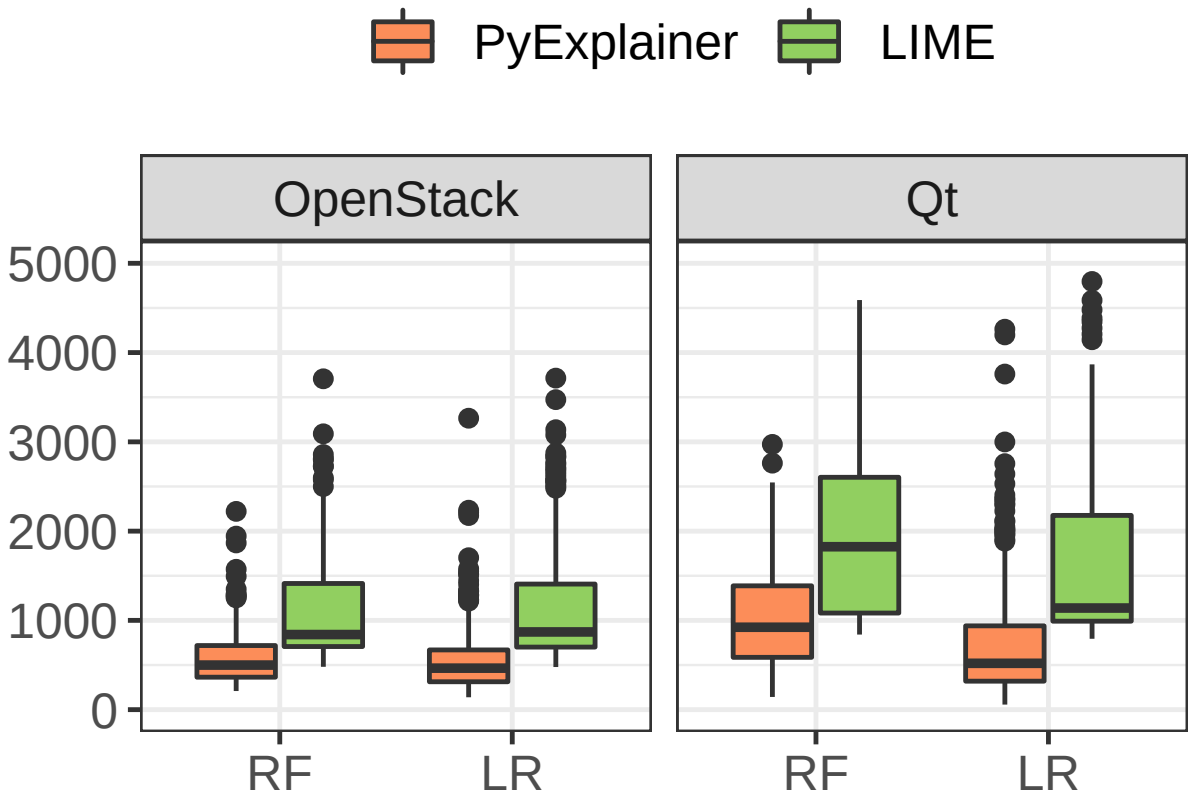


Figure 5: (RQ1) The Euclidean Distance of neighborhood instances and instances to be explained, obtained from model-agnostic techniques (i.e., PyExplainer and LIME).

Euclidean Distance of 492, while LIME achieves a median Euclidean Distance 1,825. The Wilcoxon signed-ranked test confirms that the distributions of the Euclidean Distance of our PyExplainer is statistically significantly smaller than LIME (p-value < 0.05) with a large Cliff's $|\delta|$ effect size for both classifiers and both projects.

This finding indicates that our PyExplainer produces synthetic neighbors that are more closely similar to an instance to be explained than LIME. The less similarity of synthetic neighbors generated by LIME has to do with the use of random perturbation approach. The random perturbation approach perturbs an instance to be explained by adding a value randomly drawn from a normal distribution. However, such a simple random perturbation approach is not suitable for sparse and high dimensional data like JIT datasets [37]. In particular, the random perturbation approach does not account for the characteristics of the actual JIT datasets. Thus, we found that the random perturbation approach often generates synthetic neighbors that are less similar to an instance to be explained than our PyExplainer. On the other hand, our PyExplainer generates synthetic neighbors based on the characteristics of JIT dataset using the crossover and mutation operations,

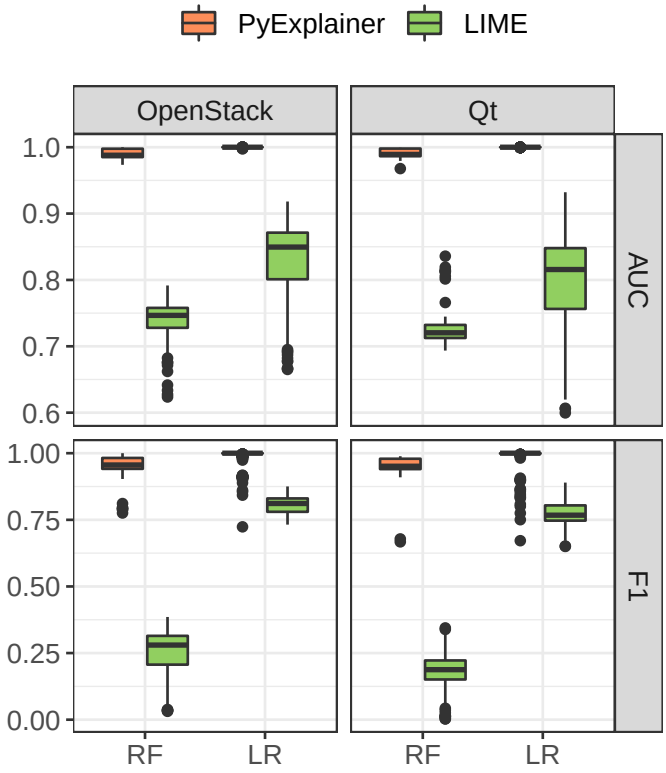


Figure 6: (RQ2) The accuracy of the local models produced by our PyExplainer and LIME in terms of AUC and F1.

producing a more accurate explanation for the predictions of JIT defect models.

(RQ2) Does our PyExplainer produce higher accuracy of local models than LIME for JIT defect models?

Approach. To address RQ2, we analyze the accuracy of the local models generated by PyExplainer and LIME. The accuracy of local models indicates how well local models can approximate (or mimic) the logic of the global models. To do so, we first obtain the predicted class (i.e., CLEAN and DEFECT) of the synthetic instances from the global JIT defect models. Then, we obtain the probability of CLEAN and DEFECT class of synthetic instances from the local models. Then, we evaluate the accuracy of the predictions between the local models and the global JIT model using two traditional performance measures, i.e., Area Under the ROC Curve (AUC) and F1. Similar to RQ1, we apply the Wilcoxon signed-rank test and the Cliff's $|\delta|$ effect size test to evaluate whether the accuracy of local models of PyExplainer are statistically significantly higher than LIME.

Results. PyExplainer produces local models that are 18%-38% more accurate than the LIME's local models. Figure 6 shows that the local models

produced by our PyExplainer achieve a median AUC of 0.99 when explaining the RF and LR JIT defect models for both Openstack and Qt. On the other hand, the local models produced by LIME achieves a median AUC of 0.75 for Openstack and 0.72 for Qt when explaining the RF JIT defect models, while achieving a median AUC of 0.85 for Openstack and 0.82 for Qt when explaining the LR JIT defect models. This indicates that our PyExplainer produces local models that are 32%-38% and 18%-24% more accurate (AUC) than LIME for both RF and LR JIT defect models. Finally, we observe a similar conclusion when using F-measure, i.e., PyExplainer produces local models that are 242.86%-413.5% and 23.46%-29.87% more accurate (F1) than LIME for both RF and LR JIT defect models. The Wilcoxon signed-ranked test confirms that the accuracy of local models produced by our PyExplainer is statistically significantly higher than the accuracy of local models produced by LIME (p-value < 0.05) with a large Cliff's $|\delta|$ effect size.

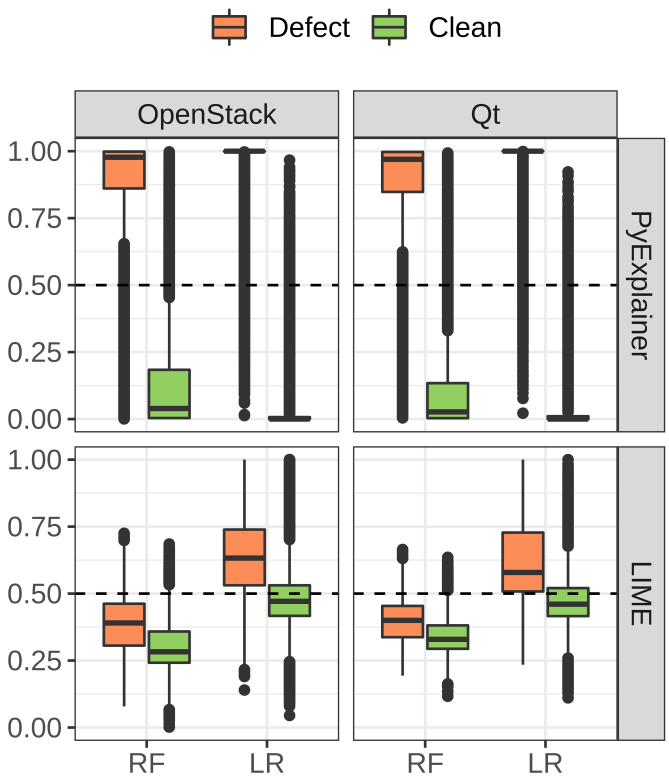


Figure 7: (RQ2) The probability (y-axis) of synthetic instances predicted by the local models of PyExplainer and LIME, when comparing to the actual class of that instances (i.e., the legend of defect and clean) from the RF and LR global JIT defect models.

The more accurate local models (i.e., high AUC) produced by our PyExplainer have to do with the quality of synthetic neighbors generated by our PyExplainer. First, our PyExplainer uses crossover and mutation techniques to generate synthetic neighbors. Thus, the synthetic neighbors are more closely similar to an

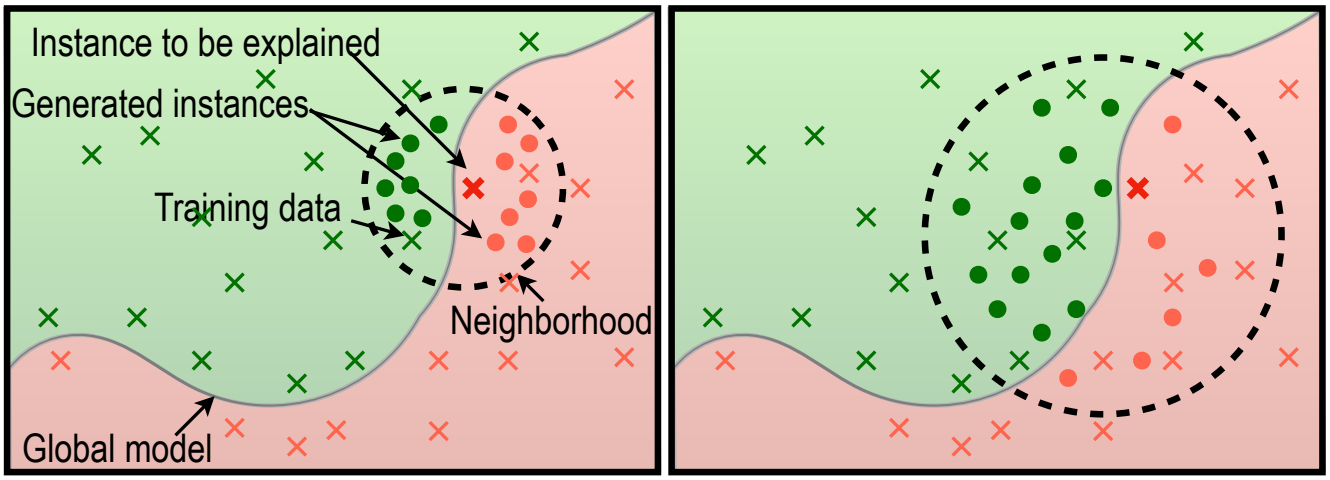


Figure 8: The characteristics of synthetic neighbors generated by PyExplainer and LIME.

instance to be explained and more similar to the actual characteristics of defect-introducing commits and clean commits from the training data. Therefore, we perform a deeper investigation to better understand the distribution of the probability of synthetic neighbours generated by the PyExplainer and LIME local models. Figure 7 shows that the median probability of defect class (0.98-1.00) and clean class (0.00-0.04) generated by the PyExplainer local models differs by 0.96-0.98. On the other hand, the median probability of defect (0.39-0.63) and clean (0.28-0.47) classes generated by the LIME local models differs by 0.11-0.16. This findings indicates that the PyExplainer local models have a higher ability to discriminate the characteristics between DEFECT and CLEAN classes, producing higher AUC than the LIME local models.

Finally, we illustrate the characteristics of synthetic neighbors generated by PyExplainer and LIME in Figure 8. In RQ1, the smaller Euclidean Distance by PyExplainer indicates that PyExplainer produces synthetic neighbors that are more similar to (1) an instance to be explained; and (2) the actual characteristics of the JIT defect datasets due to the use of crossover and mutation operations on training data. In RQ2, the higher AUC and F1 by PyExplainer indicates that our PyExplainer produces better synthetic neighbors, leading to more accurate local models than LIME. Therefore, the explanation generated by PyExplainer is more closely similar to the explanation of an instance to be explained than the explanation generated by LIME.

(RQ3) Is our PyExplainer more effective in generating explanations than LIME for JIT defect models?

Approach. To address RQ3, we analyze the explanations produced by PyExplainer and LIME using the two measures.

- *%Unique* measures the percentage of unique explanations generated by each technique. The higher percentage of unique explanations indicates that a model-agnostic technique can effectively generate a more specific (i.e., less duplicate) explanation to the instance to be explained.
- *%Consistency* measures the percentage of the defect-introducing commits in the testing data that have characteristics satisfying the rule feature in the generated explanation. The higher percentage of the consistency indicates that a model-agnostic technique can effectively generate an explanation that is consistent with the actual characteristics of defect-introducing commits.

Results. The explanations generated by our PyExplainer are 69%-98% more unique (i.e., more specific to an instance to be explained) than LIME. We find that PyExplainer can produce 100% unique explanations for all of the instances to be explained for both studied datasets. On the other hand, LIME can produce as few as 2%-4% unique explanations for OpenStack and 3%-31% unique explanations for Qt. In other words, for OpenStack, we find that as much as 72%-86% of defect-introducing commits have the same explanation, despite having different characteristics of the feature values. Similarly, for Qt, we find that as much as 53%-74% of commits have the same explanation, despite having different characteristics of the testing instances. Thus, the less duplicate explanations generated by PyExplainer indicates that PyExplainer can generate explanations that are more specific to an instance to be explained rather than LIME.

The explanations generated by our PyExplainer are 17%-54% more consistent with the actual defect-introducing commits in the testing data than LIME. Figure 9 shows that PyExplainer achieves a median consistency of 73%-75% for OpenStack and 72%-73% for Qt. On the other hand, we find that LIME achieves a median consistency of 54% for OpenStack and 18%-56% for Qt. The experiment result indicates that the explanations generated by our PyExplainer are 19%-21% and 17%-54% more consistent with the actual defect-introducing commits in the testing data than LIME for OpenStack and Qt, respectively. The Wilcoxon signed-ranked test confirms that the percentage consistency value of

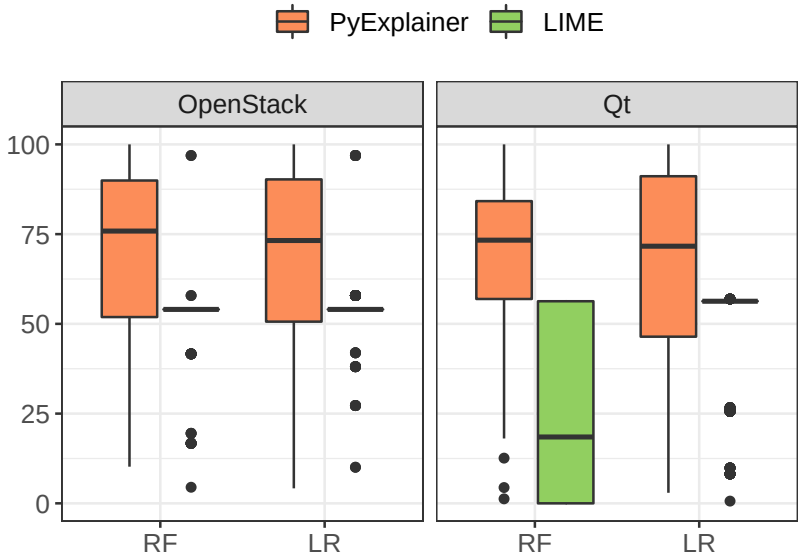


Figure 9: (RQ3) The percentage of the defect-introducing commits in the testing data that are consistent with the generated explanation.

PyExplainer is statistically significantly higher than LIME (p-value < 0.05) with a large Cliff's $|\delta|$ effect size for both JIT defect models and both studied datasets.

2.6 Discussion

In this section, we first discuss the usage scenario of how PyExplainer can be used in practice. Then, we present an analysis of the *What-If* simulation when the explanations were considered (i.e., *what if* we change this, would it reverse the predictions of the JIT defect models?). Finally, we describe the implementation details of the PyExplainer Python package.

2.6.1 A Usage Scenario

Let's consider Bob as a developer in a large-scale software project that adopts modern code review practices. Bob has his main responsibility to inspect commits that are submitted by other developers to ensure the quality of commits prior to integration into the release branch. Suppose that on average Bob spends one hour to review one commit. Hence, with his average 8 working hours per day, he can review only 8 commits per day. Given a huge number of newly arrived commits everyday (e.g., 100 commits per day), Bob does not know which commits should be reviewed first. With the use of JIT model, the list of commits can be prioritized based on the likelihood of being defect-introducing provided by the JIT defect model so that Bob can efficiently spend his limited time on the most

risky commits. However, Bob still may not be convinced by the predictions of JIT defect models, since he *does not understand why a commit is predicted as defect-introducing*. Thus, Bob may not trust the JIT defect models and may decide to ignore the predictions, resulting in suboptimal SQA resource allocation and prioritization.

With PyExplainer, Bob now better understands why a commit is predicted as defect-introducing since PyExplainer provides an explanation (i.e., which feature is the most important for a given prediction). For example, PyExplainer provides an explanation (e.g., Churn $>$ 100 $\Rightarrow$ Defect) that a commit is predicted as defect-introducing because the churn size is greater than 100. This kind of explanation could help Bob to focus on the most important aspects that are associated with the risk of being defect-introducing (i.e., considering reducing the churn size of the commit), instead of focusing on the less important aspects (e.g., inviting more reviewers). However, it remains challenging for Bob to consider which value of a feature that should be changed to mitigate the risk. In particular, given an explanation (e.g., Churn $>$ 100 $\Rightarrow$ Defect), Bob still does not know how small a Churn value should be that could reverse the prediction of JIT models from DEFECT to CLEAN. Thus, an interactive *what-if* visualization tool is needed to help Bob making better decisions of how much the Churn value that should be changed.

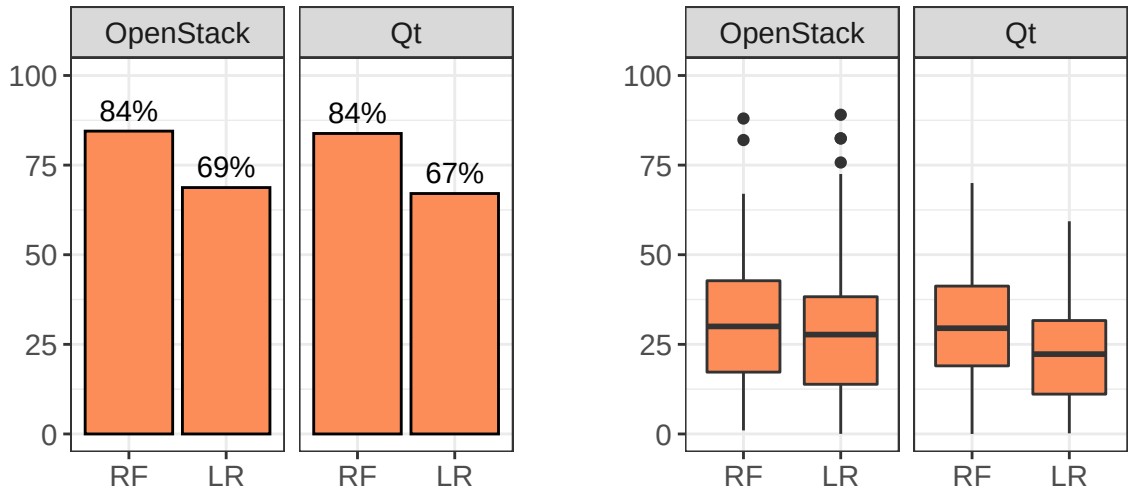
2.6.2 What-If Analysis

We conducted a *what-if* simulation based on a hypothetical scenario if the explanations of our PyExplainer were considered. In particular, we investigated *what if* we change the value of a feature guided by the explanation, would it reverse the prediction of the JIT defect model?. For example, an explanation (Churn$>$100 $\Rightarrow$ DEFECT) generated by PyExplainer means that a commit is predicted as defect-introducing since Churn is greater than 100. Thus, what if Churn was less than 100, would the prediction be reversed from DEFECT to CLEAN.

To conduct this what-if simulation, we first generate a simulated instance. The *simulated instance* is an instance where the actual value of a feature guided by the rule-based explanation was changed in the opposite direction of the comparison operator of the explanation (i.e., decrease for $>$, increase for $<$) by one SD (a standard deviation of that feature in the training data) from the rule threshold. According to the above example, the simulated instance is the modified original instance where the actual value of a feature (e.g., Churn=120) guided by the

rule-based explanation (Churn>100 $\Rightarrow$ DEFECT) was changed in the opposite direction of the comparison operator of the explanation (i.e., decrease for >) by one SD (e.g., 20) from the rule threshold (100). Thus, the Churn value of the simulated instance is 80 (i.e., 100-20). Then, we input this simulated instance to the global JIT defect model and analyze whether the simulated instance could reverse the predictions of the global JIT defect models.

We perform this what-if simulation for all the explanations generated by our PyExplainer for all of the commits in the testing dataset that are correctly predicted as defect-introducing by the JIT defect models. Then, we measure (1) *%reversed*, i.e., the percentage of the simulated instances that can reverse the prediction from DEFECT to CLEAN; and (2) *%prob_diff*, i.e., the difference between the probability of the original instance and the probability of the simulated instance.



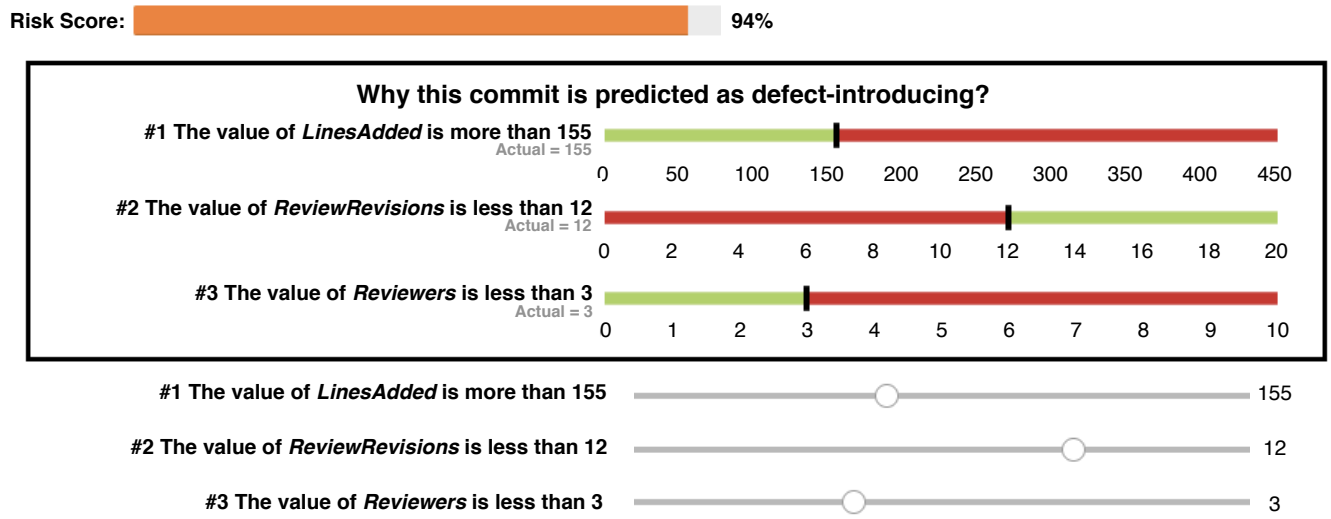
(a) The percentage of the simulated instances that can reverse the prediction from DEFECT to CLEAN.

(b) The difference between the probability of the original instance and the probability of the simulated instance.

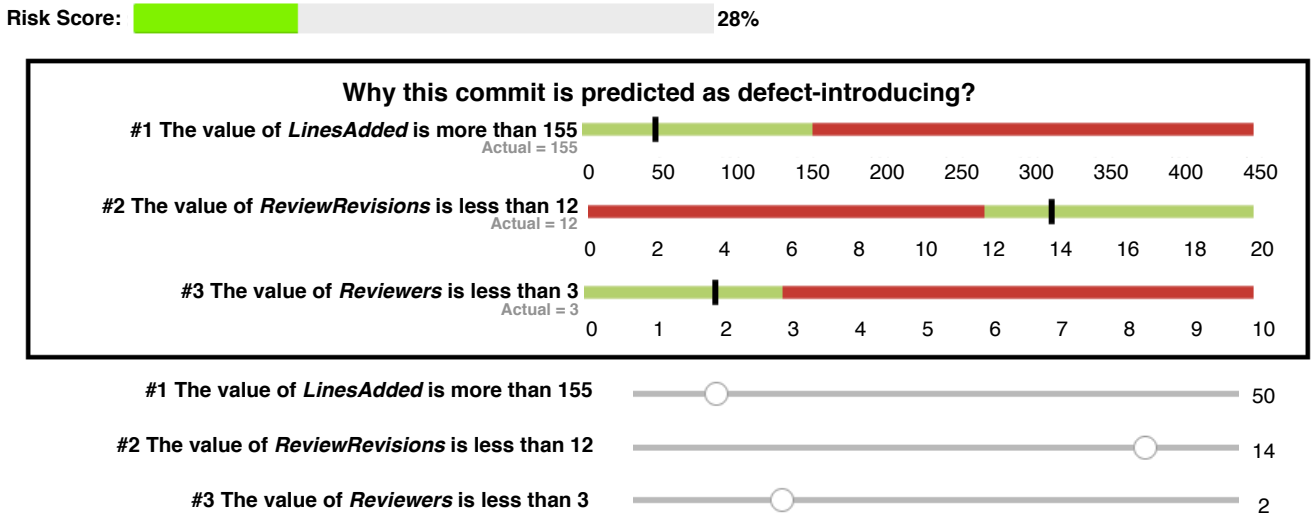
Figure 10: The results of the *what-if* analysis.

Figure 10 (a) shows that, when considering the explanations guided by our PyExplainer, 84% (RF) and 67%-69% (LR) of the simulated instances that can reverse the prediction from DEFECT to CLEAN of the global JIT defect models. Furthermore, Figure 10 (b) also shows that, after considering the explanations guided by our PyExplainer, the probability of the simulated instance is decreased by 30% for the RF models and 22% - 28% for the LR models when comparing to the probability of the original instance. This simulation highlights the importance of our PyExplainer for helping practitioners to focus on the most important aspects that are associated with the risk of being defect-introducing for a given commit, instead of focusing on the less important aspects. Nevertheless, the one SD used

in this what-if simulations is just an example, the actual changed value should be subject to the domain experts.



(a) The visual explanation of the original instance (predicted as DEFECT with a risk score of 98%).



(b) The visual explanation of the simulated instance when changing the feature values (predicted as CLEAN with a risk score of 28%).

Figure 11: The proof-of-concept visualization of our PyExplainer consists of (1) the risk score (i.e., the probability of an instance to be explained by the global JIT model); (2) the visual explanation (in the black border); and (3) the interactive what-if visualization for our PyExplainer.

2.6.3 The PyExplainer Python package

To ease the adoption of our PyExplainer by practitioners and to facilitate the replication of future research, we developed the PyExplainer Python package. In our PyExplainer Python package, we also developed a proof-of-concept of the visual explanation and the interactive what-if visualization.

The visual explanation is developed to present the rule-based explanation in a form of a bullet plot visualization with textual explanations. Figure 11 (a) shows an example of the visual explanation of an OpenStack commit (`a9a59cc`). Our visual explanation is designed to provide the following key information: (1) textual descriptions that explain why a commit is predicted as defect-introducing; (2) the actual feature values of the commit (i.e., the vertical black bars); and (3) the range of feature values associated with the risk score (i.e., the predicted probability). The green shades indicate the non-risky range values of a feature, while the red shades indicate the risky range values of a feature.

An interactive *what-if* visualization is developed to help practitioners interactively change the value of a feature, while immediately generating the new estimated risk score (i.e., the probability obtained from the JIT defect model). This visualization will allow practitioners to explore different values of a feature prior to making a decision.

Figure 11 (b) shows an example of an interactive what-if visualization for an OpenStack commit (`a9a59cc`). Through the visualization, the user can change the value of the feature (e.g., changing the value of *Lines Added* from 155 to 50, the value of *Review Revisions* from 12 to 14, and the value of *Reviewers* from 3 to 2). Then, the visualization will responsively update the predicted probability generated by the JIT defect model (e.g., from 94% to 28%).

2.6.4 Implications to Practitioners and Researchers

The contributions of this paper build an important step towards a new research area of Explainable AI for SE, by making the predictions of just-in-time defect models more explainable and actionable. A lack of explainability and actionability of software analytics has been raised by both practitioners [31, 33, 38] and researchers [32, 50, 67, 69–71]. For example, Rajapaksha *et al.* [50] proposed an approach to generate actionable suggestions (i.e., counterfactual explanations) for file-level defect prediction. Peng and Menzies [67] also proposed a TimeLIME approach (an extension of LIME model-agnostic technique) to generate actionable suggestions (i.e., defect reduction plans). However, the approaches of both Rajapaksha *et al.* [50] and Peng and Menzies [67] are designed for release-based defect prediction, which require multiple releases for training and evaluation. Thus, they are not applicable to JIT defect prediction models. On the other hand, our results show that our PyExplainer is more effective in generating explanations than LIME for the predictions of JIT defect models, while providing

an interactive what-if visualization so practitioners can make better data-informed decisions. Similar effort to other state-of-the-art model agnostic techniques (e.g., LIME [22]), we make our PyExplainer Python package publicly-available to ease the adoption by practitioners and researchers.

2.7 Threats to Validity

Threats to construct validity relates to the hyperparameter settings of Random-Forest, SMOTE, and LIME techniques when conducting our experiment [63, 72, 73]. To ensure the reprehensibility, the used parameter setting of such techniques are reported in the replication package in our GitHub repository (the replication-package branch).

Threats to internal validity relates to the randomization of our PyExplainer (i.e., the neighbour generation process). To mitigate any conclusion instability threat, we chose to generate 2,000 neighbours. After we repeated the experiment five times, the conclusion of our paper remains the same. Nevertheless, future work can explore what would be the minimum synthetic neighbours that can produce stable local explanations (i.e., the same local explanations when they are regenerated).

Threats to the external validity relates to the generalizability of our PyExplainer approach. However, our experiment only focused on the just-in-time defect prediction problem, with the limited number of the studied classification techniques, and the limited number of studied projects. Thus, other classification techniques and other projects should be explore in future work.

2.8 Summary

In this chapter, we propose PyExplainer, a novel local rule-based model-agnostic technique for explaining the predictions of JIT defect models. Through a case study of two open-source software projects, we find that our PyExplainer produces: (1) synthetic neighbours that are 41%-45% more similar to an instance to be explained; (2) 18%-38% more accurate local models; and (3) explanations that are 69%-98% more unique and 17%-54% more consistent with the actual characteristics of defect-introducing commits in the future than LIME (a state-of-the-art model-agnostic technique).

PyExplainer is designed for explaining the predictions of any classification prob-

lems. Future work should explore if our PyExplainer can be used to effectively explain the predictions of other classification problems (e.g., vulnerability prediction, code smell detection) in software engineering.

Publishing the PyExplainer Python Package. To ease the adoption of our PyExplainer by practitioners and to facilitate the replication of future research, the PyExplainer package is available in both `conda` and `pip` (Package Installer for Python). Our PyExplainer Python package also has a code coverage of 93% measured by CodeCov with an A+ quality graded by LGTM.

Figure 12: Code quality of our PyExplainer Python package.

3 JITLine: A Simpler, Better, Faster, Finer-grained Just-In-Time Defect Prediction

Chapter Overview

Recently, deep learning-based Just-In-Time (JIT) defect prediction approaches have been proposed to help developers prioritize their limited SQA resources to effectively review patches. Such approaches can predict which patches that are most likely to introduce defects after software is released. However, such approaches still cannot indicate which lines in the submitted patches that are likely to introduce defects in future. Thus, practitioners still have to spend a lot of SQA resources to locate defective lines.

In this chapter, to address the overarching research question of the thesis, we propose JITLine that incorporates the patches that are similar to the patch that JIT defect prediction models predict as defective to locate defective lines in the predicted patch. Through a case study of two open-source software projects, we find that our JITLine achieves higher performance and is faster than deep learning-based JIT defect prediction approaches [11, 12], and is more accurate than the NLP-based approach proposed by Yan *et al.* [23] in locating defective lines.

The work in this chapter appears in Chanathip Pornprasit, and Chakkrit Tantithamthavorn. “JITLine: A Simpler, Better, Faster, Finer-grained Just-In-Time Defect Prediction”. In Proceedings of MSR. 2021, pp. 369–379.

3.1 Introduction

Modern software development cycles tend to release software products in a short-term period. Such short-term software development cycles often pose critical challenges to modern Software Quality Assurance (SQA) practices. Therefore, continuous code quality tools (e.g., CI/CD, modern code review, static analysis) have been heavily adopted to early detect software defects. However, SQA teams cannot effectively inspect every commit given limited SQA resources.

Just-in-time (JIT) defect prediction [27, 74] is proposed to predict if a commit will introduce defects in the future. Such commit-level predictions are useful to help practitioners prioritize their limited SQA resources on the most risky commits during the software development process. In the past decades, several machine learning approaches are employed for developing JIT defect prediction models [7–10]. However, these approaches often rely on handcrafted commit-level features (e.g., Churn).

Recently, several deep learning approaches have been proposed for Just-In-Time defect prediction (e.g., DeepJIT [11] and CC2Vec [12]). Hoang *et al.* [12] found that their CC2Vec approach outperforms DeepJIT for Just-In-Time defect prediction. CC2Vec requires both training and unlabelled testing datasets for training CC2Vec models, assuming that all unlabelled testing datasets would be available beforehand. However, these assumptions of CC2Vec do not follow the key principles of the Just-In-Time defect prediction: (1) the predictions of the CC2Vec approach cannot be made immediately for a newly arrived commit; and (2) it is unlikely that the unlabelled testing dataset would be available beforehand when training JIT models. Thus, we perform a replication study to confirm the merit of previous experimental findings and extend their experiment by excluding testing datasets and evaluate with five additional evaluation measures.

RS1) Can we replicate the results of deep learning approaches for Just-In-Time defect prediction?
Similar to the original study [12], we are able to replicate the results of CC2Vec.

RS2) How does CC2Vec perform for Just-In-Time defect prediction after excluding testing datasets?
After excluding testing datasets when developing the JIT models, we find that the F-measure of CC2Vec is decreased by 38.5% for OpenStack and

45.7% for Qt. In addition, CC2Vec achieves a high False Alarm Rate (FAR) of 0.87 for OpenStack and 0.63 for Qt, indicating that 63%-87% clean commits are incorrectly predicted as defect-introducing. Thus, developers still waste many unnecessarily effort to inspect clean commits that are incorrectly predicted as defect-introducing.

In addition, Hoang *et al.* [12] did not compare their approach with simple JIT approaches, did not evaluate the cost-effectiveness, did not report the computational time, and cannot perform fine-grained predictions at the line level. Thus, it remains unclear about the practical value of the CC2Vec approach when considering the amount of effort that developers need to inspect.

In this chapter, we propose JITLine—a machine learning-based Just-In-Time defect prediction approach that can both predict defect-introducing commits and identify defective lines that are associated with that commit. We evaluate our JITLine approach with the state-of-the-art commit-level JIT defect prediction approaches (i.e., EARL [25], DeepJIT [11], and CC2Vec [12]) with respect to six traditional measures (i.e, AUC, F-measure, False Alarm Rate, Distance-to-Heaven, Precision, and Recall), three cost-effectiveness measures (i.e., PCI@20%LOC, Effort@20%Recall, P_{Opt}). In addition, we also compare our approach with a baseline line-level JIT defect localization by Yan [23] using four line-level effort-aware measures (i.e., Top-10 Accuracy, Recall@20%LOC, Effort@20%$Recall_{line}$, Initial False Alarm). Through a case study of 37,524 total commits that span across two large-scale open-source software projects (i.e., OpenStack and Qt), we address the following four research questions:

RQ1) Does our JITLine outperform the state-of-the-art JIT defect prediction approaches?

Our JITLine approach achieves F-measure 26%-38% higher than the state-of-the-art approaches (i.e., CC2Vec). Our JITLine achieves a False Alarm Rate (FAR) 94%-97% lower than the CC2Vec approach.

RQ2) Is our JITLine more cost-effective than the state-of-the-art JIT defect prediction approaches?

Our JITLine is 17%-51% more cost-effective than the state-of-the-art approaches in term of PCI20%Effort. In addition, our JITLine can save the amount of effort by 89%-96% to find the same number of actual defect-

introducing commits (i.e., 20% Recall) when compared to the state-of-the-art approaches.

RQ3) Is our JITLine faster than the state-of-the-art JIT defect prediction approaches?
Our JITLine is 70-100 times faster than the deep learning approaches for Just-In-Time defect prediction.

RQ4) How effective is our JITLine for prioritizing defective lines of a given defect-introducing commit?
Our JITLine approach is 133%-150% more accurate than the baseline approach by Yan *et al.* [23] for identifying actual defective lines in the top-10 recommendations. Our JITLine approach requires 17%-27% less amount of effort than the baseline approach in order to find the same amount of actual defective lines.

Contributions. The contributions of this paper are as follows:

- We conduct a replication study of the state-of-the-art deep learning approach (CC2Vec [12]) for JIT defect prediction and extend their experiment by excluding testing datasets with five evaluation measures (Section 3.3).
- We propose JITLine—a machine learning-based Just-In-Time defect prediction approach that can both predict defect-introducing commits and identify their associated defective lines (Section 3.5).
- We evaluate our JITLine approach at the commit level with the state-of-the-art JIT defect prediction approaches with respect to six traditional measures, three cost-effectiveness measures, and at the line level with four effort-aware line-level measures (Section 6.4).
- Our results show that our JITLine approach outperforms (RQ1), more cost-effective (RQ2), faster (RQ3), and more fine-grained (RQ4) than the state-of-the-art approaches.

3.2 Background

Commits created by developers are often used to describe new features, bug fixes, refactoring, etc. One commit contains three main pieces of information, i.e., a commit message, a code change, and their meta-data information (e.g.,

Table 3: The results of our replication study of CC2Vec [12] when using "train+test" and "train only" for model training.

		OpenStack						Qt					
		AUC	F1	FAR	d2h	Precision	Recall	AUC	F1	FAR	d2h	Precision	Recall
CC2Vec [Train+Test]	Original	0.81	-	-	-	-	-	0.82	-	-	-	-	-
	Ours	0.80	0.39	0.26	0.28	0.27	0.70	0.84	0.35	0.17	0.25	0.24	0.70
CC2Vec [Train Only]	Ours	0.77	0.24	0.87	0.61	0.14	0.99	0.81	0.19	0.63	0.45	0.10	0.96

churn, author name). The commit message is used to describe the semantics of the code changes, while the code change indicates changed lines (i.e., added/modified/deleted lines).

In large-scale software projects, there is a stream of commits that developers need to review and inspect. However, due to the limited SQA resources, Just-In-Time defect prediction approaches have been proposed to help developers prioritize their limited SQA resources on the most risky commits [7, 25]. Below, we discuss three state-of-the-art approaches for Just-In-Time defect prediction.

EALR [25] is an Effort-Aware JIT defect prediction method using a Logistic Regression model with traditional commit-level software metrics (e.g., churn). EALR generates a rank of defect-introducing commits by considering the amount of inspection effort—i.e., the predicted probability is normalized by the commit size (i.e., churn). However, such techniques often rely on handcrafted feature engineering.

DeepJIT [11] is an end-to-end deep learning framework for Just-in-Time defect prediction. DeepJIT automatically generates features using a Convolutional Neural Network (CNN) architecture. Generally, DeepJIT takes the commit message and the code change as an input into two CNN models in order to generate a vector representation—i.e, one CNN for generating commit message vectors and another CNN for generating code changes vectors. Finally, the concatenation of both the commit message vector and the code change vector is input into the fully-connected layer to generate the probability of defect-introducing commit.

CC2Vec [12] is an approach to learn the distributed representation of commit. Traditionally, one commit has a hierarchical structure–i.e., one commit consists of changed files, one change file consists of changed hunks, one change hunk consists of changed lines, one changed line consists of changed tokens. Unlike DeepJIT that ignores the information about the hierarchical structure of code commits, CC2Vec has been proposed to automatically learn the hierarchical

structure of code commits using a Hierarchical Attention Network (HAN) architecture. The goal of CC2Vec is to learn the relationship between the actual code changes and the semantic of that code changes (i.e., the first line of commit messages). Then, in the feature extraction layer, HAN is used to build vector representations of changed lines; these vectors are then used to construct vector representations of hunks; and then these vectors are aggregated to construct the embedding vector of the removed or added code. Then, the embedding vectors of the removed code and added code is input into a fully-connected layer to generate a vector that represents the code change.

Recently, Hoang *et al.* has shown that the combination of CC2Vec and DeepJIT outperforms the stand-alone DeepJIT approach. In particular, they used CC2Vec to generate a vector representation of code changes. Then, such code changes vector is concatenated with the commit message vectors and the code change vectors that are generated by DeepJIT to generate a final vector representation. Finally, the concatenation vector is input into the fully-connected layer to generate the probability of defect-introducing commit.

3.3 A Replication Study of the State-of-the-art Deep Learning Approach for JIT Defect Prediction

In this section, we present the motivation, approach, and results of our replication study (RS) of CC2Vec for Just-In-Time defect prediction.

Motivation. One of the key principles of *Just-In-Time* defect prediction models is to *generate predictions as soon as possible* for a newly arrived commit. Let's consider T_1 as the present (see Figure 13), the whole historical data will be used for training a JIT model in order to immediately generate a prediction of a newly arrived commit. However, CC2Vec requires both training and unlabelled testing datasets for training CC2Vec models (i.e., the periods of T_0-T_1 and T_1-T_2), assuming that all unlabelled testing datasets would be available beforehand. In particular, Hoang *et al.* (Section 3.3.3 of the original study [12]) stated that *"CC2Vec is first used to learn distributed representations of the code changes in the whole dataset. All patches from the training and testing dataset are used since the log messages of the testing dataset are not part of the predictions of the task"*. This indicates that the unlabelled testing dataset needs to be available beforehand for training CC2Vec models. However, these assumptions of CC2Vec do not follow the key principles of the Just-In-Time defect prediction: (1) the predictions of the CC2Vec approach cannot be made immediately for a newly arrived commit; and

(2) it is unlikely that the unlabelled testing dataset would be available beforehand when training JIT models. Thus, it remains unclear what the performance of CC2Vec for Just-In-Time defect prediction is after considering the key principle of Just-In-Time defect prediction (i.e., excluding testing dataset for model training). In addition, several other performance measures (e.g., F-measure) have not been evaluated in the original study. Thus, we (RS1) perform a replication study to confirm the merit of previous experimental findings and (RS2) extend their experiment by excluding testing datasets and evaluate with five additional evaluation measures.

(RS1) Can we replicate the results of deep learning approaches for Just-In-Time defect prediction?

Approach. To address RS1, we first download the replication package of Hoang *et al.* [12]. We carefully study the replication package to understand all details. Then, we execute the source code followed by the instructions and datasets provided by Hoang *et al.* [12]. Finally, we compute a relative percentage between our results and the original paper as follows: $\% = (\frac{\text{ours}-\text{original}}{\text{original}}) \times 100\%$

Results. **Similar to the original study [12], we are able to replicate the results of CC2Vec.** Table 3 (see the green cells) shows that, in our experiment, CC2Vec achieves an AUC of 0.80 for OpenStack and 0.84 for Qt, while the original paper reported an AUC of 0.81 for OpenStack and 0.82 for Qt. Our results are only 1%-2% different when compared to the original paper. This finding confirms that the results of CC2Vec are replicable for Just-In-Time defect prediction.

(RS2) How does CC2Vec perform for Just-In-Time defect prediction after excluding testing datasets?

Approach. To address RS2, we repeat the experiment of Hoang *et al.* [12] in two settings—i.e., the original experiment with training and testing datasets and our experiment with training datasets only. In addition, we extend their experiment by evaluating the CC2Vec approach using five additional evaluation measures (i.e., F-measure, False Alarm Rate, Distance-to-Heaven, Precision, and Recall).

Results. **After excluding testing datasets when developing the JIT models, we find that the F-measure of CC2Vec is decreased by 38.5%(0.35→0.19) for OpenStack. and 45.7%(0.39→0.24) for Qt.** Table 3 (see the red cells) shows the result between two experimental settings (i.e., training+testing vs. training

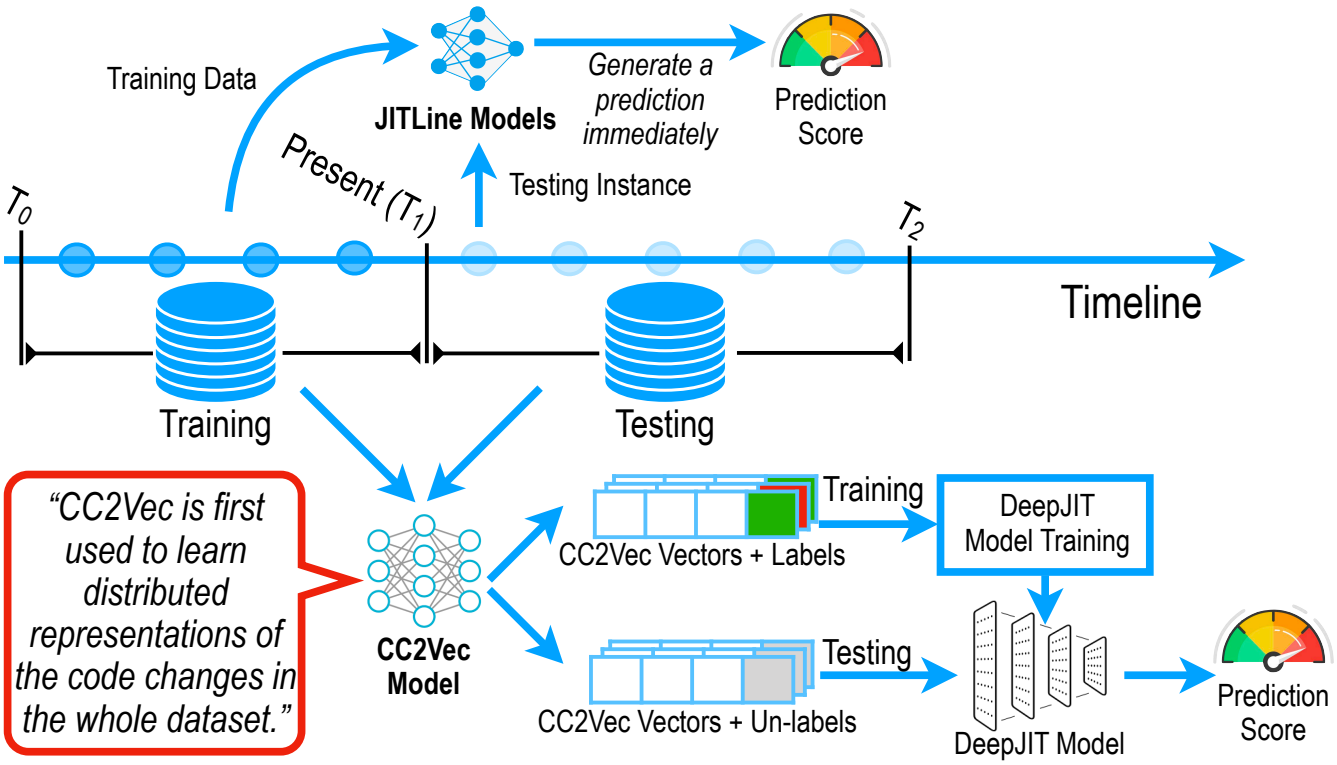


Figure 13: The comparison between the workflow of JITLine that can immediately generate predictions and the workflow of CC2Vec+DeepJIT [12] which requires testing dataset to be available beforehand for training CC2Vec+DeepJIT models.

only) with respect to AUC, F1, FAR, d2h, Precision and Recall. We find that the values of several performance measures (i.e, AUC, F-measure, FAR, d2h) are negatively impacted by the exclusion of the testing datasets. We find that AUC is decreased by 3.9% for OpenStack and 3.7% for Qt, while False Alarm Rates (FAR) are increased by 234.62% for OpenStack and 270.59% for Qt. Similarly, the d2h value is increased by 126.92% for Openstack and 80% for Qt. The higher FAR and d2h of CC2Vec has to do with the substantial increasing Recall to 0.99 for OpenStack and 0.96 for Qt—i.e., CC2Vec predicts most of the commits as defect-introducing (higher Recall), but many of the predictions are incorrect (higher FAR, less Precision). These findings indicate that the exclusion of testing datasets in model training has a large negative impact on the performance of CC2Vec (i.e., producing higher False Alarm Rates). Thus, developers have to waste unnecessarily effort on inspecting clean commits that are incorrectly predicted as defect-introducing.

3.4 Related Work and Research Questions

In this section, we discuss the following four main limitations of prior studies with respect to the literature in order to motivate our approach and research questions.

First, several traditional machine learning-based JIT approaches have not been compared with the deep learning approaches for JIT defect prediction. Recently, researchers found that several simple approaches often outperform deep learning approaches in SE tasks. For example, Hellendoorn [75], Fu and Menzies [76], Liu *et al.* [77]. Menzies *et al.* [78] suggested that researchers should explore simple and fast approaches before applying deep learning approaches on SE tasks. However, Hoang *et al.* [12] did not compared their CC2Vec approach with other simple approaches (e.g., logistic regression and random forest). Therefore, we wish to investigate if our approach outperforms the deep learning approaches for Just-In-Time defect prediction.

Second, the cost-effectiveness of deep learning approaches for JIT defect prediction has not been investigated. Prior work pointed out that different code changes often require different amount of code inspection effort [79, 80]—i.e., large code changes often require a high amount of code inspection effort. However, Hoang *et al.* [12] did not investigate the cost-effectiveness of their CC2Vec approach. In addition, the CC2Vec approach does not take into consideration the effort required to inspect code changes when prioritizing defect-introducing commits. Therefore, we wish to investigate if our approach is more cost-effective than the deep learning approaches for Just-In-Time defect prediction.

Third, the computational time of deep learning approaches JIT defect prediction has not been investigated. Several researchers raised concerns that deep learning approaches are often complex and very expensive in terms of GPU costs/CPU time. For example, Jiang *et al.* [81]'s approach requires 38 hours for training their deep learning models on NVIDIA GeForce GTX 1070. Menzies *et al.* [78] found that a simple approach that is 500+ times faster achieves similar performance to deep learning approaches. Therefore, we wish to investigate if our approach is faster than the deep learning approaches for Just-In-Time defect prediction.

Finally, there exists no machine learning approaches for fine-grained Just-In-Time defect prediction at line level. Recently, Pascarella *et al.* [82] proposed a fine-grained JIT defect prediction model which based on handcrafted

features to prioritize which changed files in a commit should be review first. However, this approach cannot identify defective lines of the changed files. Recently, Yan *et al.* [23] proposed a fine-grained JIT defect localization at the line level to help developers to locate and address defects using less effort. Yan *et al.* [23] proposed a two-phase approach—i.e., the ML model trained on software metrics (e.g., #added_lines) is first used to identify which commits are the most risky, then the N-gram model trained on textual features is finally used to localise the riskiest lines. On the other hand, a recent work by Wattanakriengkrai *et al.* [45] pointed out that a machine learning approach outperforms the n-gram approach. However, their experiment focused solely on file-level defect prediction—not Just-In-Time defect prediction. Therefore, we wish to investigate if our approach is more effective than the two-phase approach for Just-In-Time defect prediction.

Considering the limitations yet high impact of prior work, we propose JITLine—a machine learning-based Just-In-Time defect prediction approach that can predict both defect-introducing commits and their associated defective lines. Then, we formulate the following research questions:

RQ1) Does our JITLine outperform the state-of-the-art JIT defect prediction approaches?

RQ2) Is our JITLine more cost-effective than the state-of-the-art JIT defect prediction approaches?

RQ3) Is our JITLine faster than the state-of-the-art JIT defect prediction approaches?

RQ4) How effective is our JITLine for prioritizing defective lines of a given defect-introducing commit?

3.5 JITLine: A JIT Defect Prediction Approach at the Commit and Line Levels

In this section, we present the implementation of our JITLine approach. The goal of our JITLine approach is to predict defect-introducing commits and identify lines that are associated with that defect-introducing commit (i.e., defective lines). The underlying intuition of our approach is that code tokens that frequently appeared in defect-introducing commits in the past are likely to be fixed in the future.

Overview. Our approach begins with extracting source code tokens of code changes as features (i.e., token features). Since our JIT defect datasets are

highly imbalanced (i.e., 8%-13% defective ratio), we apply a SMOTE technique that is optimized by a Differential Evolution (DE) algorithm to handle the class imbalance issue on a training dataset. Then, we build commit-level JIT defect prediction model using the rebalanced training dataset. Next, we generate a prediction for each commit in a testing dataset. After that, we normalize the prediction score by the amount of code changes (i.e., churn) in order to consider the inspection effort when generating the ranking of defect-introducing commits. For each commit in the testing dataset, we extract the importance score of each token features using a state-of-the-art model-agnostic technique, i.e., Local Interpretable Model-Agnostic Explanations (LIME). Finally, we rank defective lines that are associated with a given commit based on the LIME's importance scores. We describe each step in details below.

(Step 1) Extracting Bag-of-Tokens Features. Following the underlying intuition of our approach, we represent each commit using Bag-of-Tokens features (i.e., the frequency of each code token in a commit). To do so, for each commit, we first perform a code tokenization step to break each changed line into separate tokens. Then, we parse its removed lines or added lines into a sequence of tokens. As suggested by Rahman *et al.* [83], removing these non-alphanumeric characters will ensure that the analyzed code tokens will not be artificially repetitive. Thus, we apply a set of regular expressions to remove non-alphanumeric characters such as semi-colon (;) and equal sign (=). We also replace the numeric literal and string literal with a special token (i.e., `<NUM>` and `<STR>` respectively) to reduce the vocabulary size. Then, we extract the frequency of code tokens for each commit using the `Countvectorize` function of the Scikit-Learn Python library. We neither perform lowercase, stemming, nor lemmatization (i.e., a technique to reduce inflectional forms) on our extracted tokens, since the programming language of our studied systems is case-sensitive. Otherwise, the meaning of code tokens may be discarded if stemming and lemmatization are applied.

(Step 2) Handling class imbalance using an Optimized SMOTE technique. Since our JIT defect datasets are highly imbalanced (i.e., 8%-13% defective ratio), we apply a SMOTE technique that is optimized by a Differential Evolution (DE) algorithm to handle the class imbalance issue on a training dataset. The training dataset is splitted into a new training set and a validation set. The new training set is used to train DE+SMOTE, while the validation set is used to select the best hyper-parameter settings. We select the SMOTE technique, as prior studies have shown that the SMOTE technique outperforms other class rebal-

ancing techniques [62, 63].

The SMOTE technique starts with a set of minority class (i.e., defect-introducing commits). For each of the minority class of the training datasets, SMOTE calculates the k-nearest neighbors. Then, SMOTE selects N instances of the majority class (i.e., clean commits) based on the smallest magnitude of the euclidean distances that are obtained from the k-nearest neighbors. Finally, SMOTE combines the synthetic oversampling of the minority defect-introducing commits with the undersampling the majority clean commits. We use the implementation of `SMOTE` function provided by the `Imbalanced-Learn` Python library [65]. However, prior studies pointed out that the SMOTE technique involves many parameters settings (e.g., k the number of neighbors, m the number of synthetic examples to create, r the power parameter for the Minkowski distance metric), which often impact the accuracy of prediction models [62, 63, 72, 84].

To ensure that we achieve the best performance of the SMOTE algorithm, we optimize the SMOTE technique using a Differential Evolution (DE) algorithm (as suggested by Agrawal *et al.* [62] and Tantithamthavorn *et al.* [63]). DE [85] is an evolutionary-based optimization technique, which is based on a differential equation concept. Unlike a Genetic Algorithm technique that uses crossover as search mechanisms, a DE technique uses mutation as a search mechanism. First, DE generates an initial population of candidate setting of SMOTE's k nearest neighbors with a range value of 1-20. Then, DE generates new candidates by adding a weighted difference between two population members to the third member based on a crossover probability parameter. Finally, DE keeps the best candidate SMOTE's parameter setting that is evaluated by a fitness function of maximizing an AUC value for the next generation. We use the implementation of the differential evolution algorithm provided by Scipy Python library [86]. As suggested by Agrawal *et al.* [62], we set the population size to 10, the mutation power to 0.7 and a crossover probability (or `recombination` parameter in Scipy) to 0.3.

(Step 3) Building commit-level JIT defect prediction models. We build a commit-level JIT defect model using both the Bag-of-Tokens features from Step 1 and the commit-level metrics from McIntosh and Kamei [30]. The details of commit-level metrics are provided in the replication package. Prior work found that different classification techniques often produce different performance measures. Thus, we conduct an experiment on different classification techniques. We consider the following well-known classification techniques [62, 63, 72, 87],

i.e., Random Forest (RF), Logistic Regression (LR), Support Vector Machine (SVM), k-Nearest Neighbours (kNN), and AdaBoost. For each project, we build the JIT model using the implementation provided by Python Scikit-Learn package. We find that LR, kNN, and SVM cannot be built with the Qt project due to the high-dimensional feature space, and the model training time for such models (which takes few hours) is considerably larger than RF (which takes few minutes). Therefore, we only select the Random Forest classification technique for our study. After we experiment with different parameter settings of trees (a range of 50 to 1,000), we find that our approach is not sensitive to the parameter setting of random forest. Thus, we set the number of tress of random forest to 300.

(Step 4) Computing a defect density of each commit. We then generate the prediction probability for each commit in the testing dataset using the `predict_proba` function provided by the Scikit-learn Python library. Then, we compute the defect density as the probability score normalized by the total changed lines of code of that commit $(\frac{\mathsf{Y}(m)}{\#\mathsf{LOC}(c)})$. The use of defect density is suggested by prior studies [25, 79] who argued that the cost of applying quality assurance activities may not be the same for each code changes. In other words, a prediction model that prioritizes the largest commit as most defect prone would have a very high recall, i.e., those commits likely contain the majority of defects, yet inspecting all those commits would take a considerable amount of time. In contrast, a model that recommends slightly less defect-prone commits that are smaller to inspect would be more cost-effective [25].

(Step 5) Generating a ranking of defective lines for a given commit. In our studied projects, we found that the average size of the commit varies from 73 to 140 changed lines, but the average ratio of actual defective lines is as low as 51%-53%. Thus, developers still spend unnecessarily effort on locating actual defective lines of that commit [45]. To address this challenge, we propose to generate a ranking of defective lines for a given commit. For each commit, we compute the importance score of token features using a Local Interpretable Model-agnostic Explanations (LIME) technique. LIME [22] is a model-agnostic technique that aims to mimic the behavior of the predictions of the defect model by explaining the individual predictions. Given a commit-level JIT defect prediction model and a commit in the testing dataset, LIME performs the following steps:

1. Generate neighbor instances of a test instance x. LIME randomly gener-

Table 4: The statistics of our studied datasets.

	#Commits	%Defect-Introducing Commits	#Unique Tokens	Avg. of Commit Size	Avg. of %Defective Lines
Openstack	12,374	13%	32K	73 LOC	53%
Qt	25,150	8%	81K	140 LOC	51%

ates n synthetic instances surrounding the test instance x using a random perturbation method with an exponential kernel function on cosine distance.

2. Generate labels of the neighbors using a commit-level JIT defect prediction model. LIME uses the commit-level JIT defect prediction model to generate the predictions of the neighbor instances.
3. Generates local explanations from the generated neighbors. LIME builds a local sparse linear regression model (K-Lasso) using the randomly generated instances and their generated predictions from the commit-level defect model. The coefficients of the K-Lasso model indicate the importance score of each feature on the prediction of a test instance according to the K-Lasso model.

The LIME's importance score of each token feature ranges from -1 to 1. A positive LIME score of a token feature ($0 < e \leq 1$) indicates that the feature has a positive impact on the estimated probability of the test instance (i.e., **risky tokens**). On the other hand, a negative LIME score of a token feature ($-1 \leq e < 0$) indicates that the token feature has a negative impact on the estimated probability (i.e., **non-risky tokens**). Once the importance score of each token is computed, we generate the ranking of defect-prone lines using the summation of the importance score for all tokens that appear in that line. We use the implementation of LIME provided by the `lime` Python package.

3.6 Experimental Setup and Results

In this section, we describe the studied datasets and present the experimental results with respect to our four research questions.

Studied Datasets. In this paper, we select the dataset of McIntosh and Kamei [30] due to the following reasons. First, we would like to establish a fair comparison, using the same training and testing datasets with previous work [11, 12],

where this dataset was used. Second, we would like to ensure that our results rely on high quality datasets. Recently, researchers raised concerns that the SZZ algorithm [53] may produce many false positives and false negatives [88]. However, the datasets of McIntosh and Kamei [30] have been manually verified through many filtering steps (e.g., ignore comment updates, ignore white space/indentation changes, remove mislabelled defect-introducing commits). Finally, we select the datasets of McIntosh and Kamei [30] with two open-source software systems, i.e., OpenStack and Qt. Openstack is an opensource software for cloud infrastructure service. Qt is a cross-platform application development framework written in C++. Table 15 presents the statistics of the studied datasets.

Below, we present the approach and the results with respect to our four research questions.

(RQ1) Does our JITLine outperform the state-of-the-art JIT defect prediction approaches?

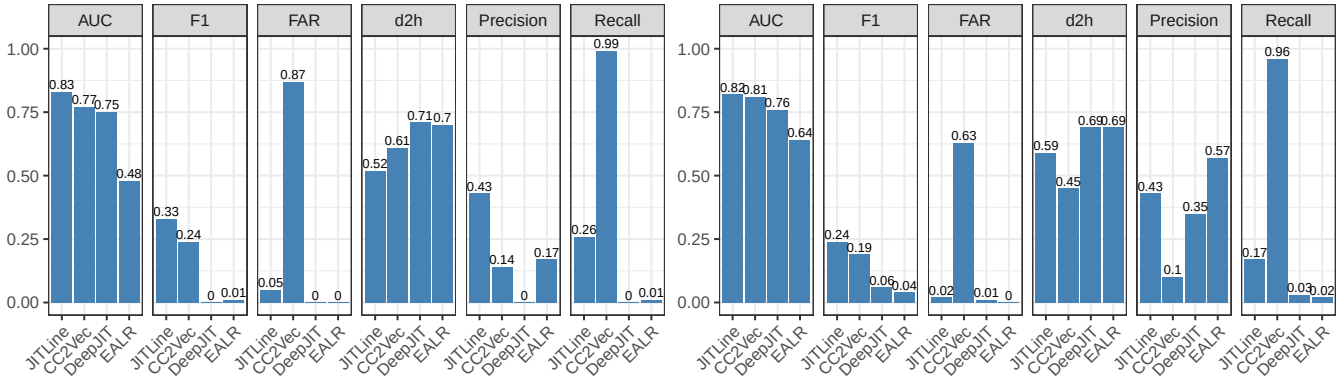


Figure 14: (RQ1) The evaluation result of our JITLine approach compared with the state-of-the-art approaches for Just-In-Time defect prediction (i.e., CC2Vec(Train only), DeepJIT, and EALR).

Approach. To answer this RQ, we evaluate our JITLine using the same training/testing datasets as prior studies [11, 12, 30] to establish a fair comparison. For training, we use 11,043 commits for OpenStack and 22,579 commits for Qt. For testing, we use 1,331 commits for OpenStack and 2,571 for Qt. Since our JIT defect datasets are time-wise, we do not perform cross-validation to avoid the use of testing data in the training data [89]. Then, we compare our JITLine with the following three state-of-the-art JIT defect prediction approaches (i.e., EARL, DeepJIT, CC2Vec). The details of the state-of-the-art approach is provided in Section 6.2. Finally, we evaluate these approaches using the following six traditional evaluation measures [11, 12, 45, 62].

1. AUC is an Area Under the ROC Curve (i.e., the true positive rate and the false positive rate). AUC values range from 0 to 1, with a value of 1 indicates perfect discrimination, while a value of 0.5 indicates random guessing.
2. F-measure is a harmonic mean of precision and recall, which is computed as $\frac{2\times \text{Precision}\times \text{Recall}}{\text{Precision}+\text{Recall}}$. We use the probability threshold of 0.5 for calculating precision and recall.
3. False Alarm Rate (FAR) [87] measures the ratio of incorrectly predicted defect-introducing commits and the number of actual clean commits $\frac{\text{FP}}{\text{FP}+\text{TN}}$. The lower the FAR value is, the fewer the incorrectly predicted defect-introducing commits that developers need to review. In other words, a low FAR value indicates that developers will spend less effort on reviewing the incorrectly predicted defect-introducing commits.
4. Distance-to-Heaven (d2h) [87] is a root mean square of recall and FAR values, which can be computed as $\sqrt{\frac{(1-\text{Recall})^2+(0-\text{FAR})^2}{2}}$. A d2h value of 0 indicates that an approach can correctly predict all defect-introducing commits without any false positive. A high d2h value indicates that an approach is far from perfect, e.g., achieving a high recall value but also have high FAR value and vice versa.
5. Precision measures the ability of an approach to correctly predict defect-introducing commits, which can be calculated as follows: $\text{Precision} = \frac{\text{TP}}{\text{TP}+\text{FP}}$. The higher precision, the better model to correctly predict defect-introducing commits.
6. Recall measures the ability to correctly retrieve defect-introducing commits when making a prediction. The calculation of this measure is $\text{Recall} = \frac{\text{TP}}{\text{TP}+\text{FN}}$. High recall indicates that the model can obtain a lot of defect-introducing commits during prediction.

Results. **Our JITLine approach achieves an AUC 28%-73% higher and an F-measure 26%-38% higher than the state-of-the-art approaches (i.e., CC2Vec).** Figure 14 presents the experimental results of our approach and the state-of-the-art approaches with respect to six evaluation measures, i.e., AUC, FAR, d2h, precision, and recall for Openstack and Qt. We find that our JITLine approach achieves the highest AUC value of 0.83 for Openstack and 0.82 for Qt, which is 1%-8% higher than CC2Vec, 8%-10% higher than DeepJIT, and 28%-73% higher than EALR. We also find that our JITLine approach achieves the highest

F-measure value of 0.33 for Openstack and 0.24 for Qt, which are 26%-38% higher than CC2Vec, 300%-3,300% higher than DeepJIT, and 500%-3,200% higher than EALR. This finding indicates that our approach outperforms the state-of-the-art approaches in terms of AUC and F-measure.

Our JITLine approach achieves a False Alarm Rate (FAR) 94%-97% lower than the CC2Vec approach. We find that our JITLine approach achieves a False Alarm Rate (FAR) of 0.05 for Openstack and 0.02 for Qt, which is similar to DeepJIT (FAR=0.01) and EALR (FAR=0). Similarly, our JITLine approach also achieves a d2h of 0.52 for OpenStack and 0.59 for Qt, which is lower than the state-of-the-art approaches (i.e., DeepJIT and EALR). However, we observe that the d2h of our approach for Qt project is higher than the CC2Vec approach. For Qt project, we find that the lower d2h value of the CC2Vec approach has to do with the high recall value of 0.96—i.e., the CC2Vec approach predicts most of the commits as defect-introducing, but 63%-87% of them are incorrect (i.e., many of them are false positives)— indicating that developers may spend unnecessarily effort to inspect actual clean commits that are incorrectly predicted as defect-introducing commits when the CC2Vec approach was used. On the other hand, the high d2h value of our approach has to do with the low recall of 0.16, but our approach achieves a low FAR of 0.02, indicating that the predictions from our JIT-Line approach is less likely to predict actual clean commits as defect-introducing. After considering both the ability of identifying defect-introducing commits (i.e., Recall) and the additional costs (i.e., FAR), our approach still outperforms state-of-the-art approaches (i.e., CC2Vec (only for OpenStack), DeepJIT, and EALR).

(RQ2) Is our JITLine more cost-effective than the state-of-the-art JIT defect prediction approaches?

Approach. To answer this RQ, we evaluate our JITLine and compare with the four state-of-the-art JIT defect prediction approaches (as mentioned in RQ1) using the following cost-effective measures [25, 79, 80, 87, 90]:

1. PCI@20%LOC measures the proportion of actual defect-introducing commits that can be found given a fixed amount of effort, i.e., the Top 20% LOC of the whole project. A high value of PCI@20%LOC indicates that an approach can rank many actual defect-introducing commits so developers will spend less effort to find actual defect-introducing commits.
2. Effort@20%Recall measures the amount of effort (measured as LOC) that

developers have to spend to find the actual 20% defect-introducing commits divided by the total changed LOC of the whole testing dataset. A low value of Effort@20%Recall indicates that the developers will spend a little amount of effort to find the 20% actual defect-introducing commits.

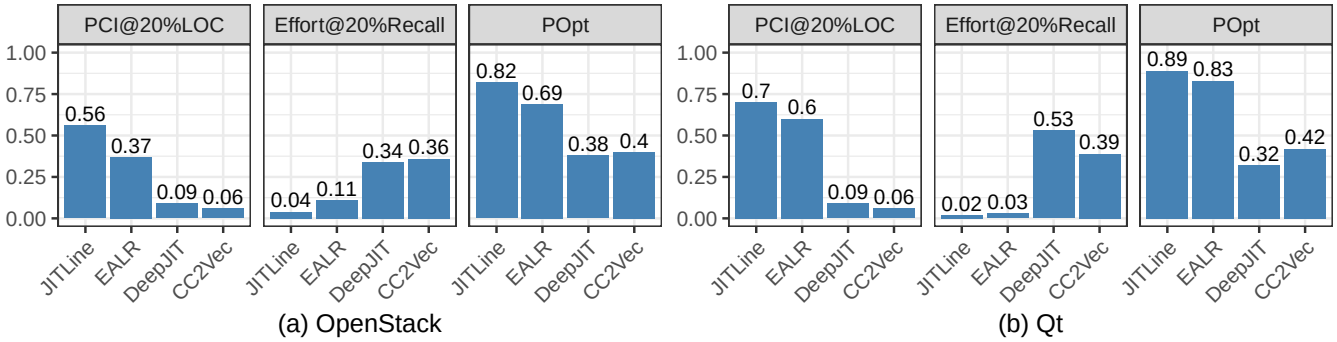


Figure 15: (RQ2) The cost-effectiveness of our JITLine approach compared to the state-of-the-art approaches for Just-In-Time defect prediction with respect to PCI@20%Recall, Effort@20%Recall, and P_{Opt}.

3. P_{opt} is defined as 1-Δ_{opt}, where Δ_{opt} is the area of the effort-based (i.e., churn) cumulative lift chart between an optimal model and a prediction model. The effort-based (i.e., churn) cumulative lift chart is the relationship between the cumulative percentage of defect-introducing commits from a prediction model (y-axis) and the cumulative percentage of the inspection effort (x-axis). Similar to prior studies [79, 87, 90], we use the normalized version of P_{opt}, which is defined as $1 - \frac{\text{Area(Optimal)} - \text{Area(Our)}}{\text{Area(Optimal)} - \text{Area(Worst)}}$. For the *optimal* model and the *worst* model, all commits are ranked by the actual defect density in descending and ascending order, respectively. For *our* model, all commits are ranked by the estimated defect density $(\frac{\text{Y}(m)}{\#\text{LOC}(c)})$ in descending order.

Results. **Our JITLine approach is 17%-51% more cost-effective than the state-of-the-art approaches in term of PCI@20%LOC.** Figure 15 presents the cost-effectiveness of our JITLine approach compared to the state-of-the-art approaches for Just-In-Time defect prediction with respect to the PCI@20%LOC, Effort@20%Recall and P_{opt} measures. We find that our JITLine approach is more cost-effective than the state-of-the-art approaches for three cost-effectiveness measures. We find that our JITLine achieves a PCI@20%LOC of 0.56 for Openstack and 0.70 for Qt, while the state-of-the-art achieves a PCI@20%LOC of 0.06-0.37 for OpenStack and 0.06-0.60 for Qt. This finding indicates that given a fixed amount of inspection effort at 20%LOC, our JITLine approach can correctly predict 17%-51% higher number of actual defect-introducing commits than the state-of-the-art approaches.

Table 5: (RQ3) The average CPU and GPU computational time (minutes±95% Confidence Interval) of the model training of JIT defect prediction approaches after repeating the experiment 5 times.

	CPU		GPU	
	Openstack	**Qt**	**Openstack**	**Qt**
JITLine	36±1 secs	175±1 secs	-	-
DeepJIT	70±7 mins	143±7 mins	2±0.01 mins	5±0.01 mins
CC2Vec	146±16 mins	300±6 mins	13±0.05 mins	30±0.10 mins
EARL	8±1 secs	97±1 secs	-	-

Our JITLine approach can save the amount of effort by 89%-96% to find the same number of actual defect-introducing commits (i.e., 20% Recall) when compared to the state-of-the-art approaches. Our JITLine approach achieves an Effort@20%Recall of 0.04 for Openstack and and 0.02 Qt, while the state-of-the-art approaches achieve an Effort@20%Recall of 0.11-0.36 for Openstack, and 0.03-0.53 for Qt. Similarly, our JITLine approach achieves a P_{opt} of 0.82 for OpenStack and 0.89 for Qt, which is 116% and 178% higher than the state-of-the-art approaches for OpenStack and Qt, respectively. In particular, our P_{opt} is 7% to 19% higher than EALR, 116% to 178% higher than DeepJIT, and 105% to 112% higher than CC2Vec. This finding suggests that, to find the same amount of actual defect-introducing commits, our JITLine approach can reduced the amount of effort by 85% and 96% when compared to the state-of-the-art approaches, which may provide the best return on investment.

(RQ3) Is our JITLine faster than the state-of-the-art JIT defect prediction approaches?

Approach. To answer this RQ, we measure the CPU computational time of the model training of our approach, and the CPU and GPU computational time of the model training of deep learning approaches (i.e., DeepJIT and CC2Vec). For our approach, we set `n_jobs` argument of the `RandomForestClassifier` function of Scikit-Learn library to -1 to ensure that all available CPU cores are used in parallel. For the deep learning baselines, we use `cpu` function provided by the Pytorch deep learning library to ensure that all available CPU cores are used in parallel. We perform the experiment using the following equipment: AMD Ryzen 9 5950X 16 Cores/32 Threads Processor, RAM 64GB, NVIDIA GeForce RTX 2080 Ti 11GB. To ensure that our measurement is accurate and strictly controlled, we reserve the computing resources and ensure that the resources are idle with

no other running tasks. To combat the randomization bias, we repeat the experiment 5 times.

Results. **Our JITLine approach is 70-100 times faster than the deep learning approaches for Just-In-Time defect prediction.** Table 5 presents the average CPU and GPU computational time (minutes) of the model training of JIT defect prediction approaches after repeating the experiment 5 times. We find that the model training time of our JITLine approach takes approximately 1-3 minutes, while the model training time of the deep learning approaches for Just-In-Time defect prediction require 1-5 hours (70 to 300 minutes). Given the same running cost (on CPU), this finding suggests that our approach is more cost-efficient than the deep learning approaches.

The computation time of the deep learning approaches can be accelerated by using a high-end GPU hardware. However, we find that the model training time of the deep learning approaches on the GPU device is relatively faster than using the CPU hardware with an additional GPU cost. Nevertheless, the model training time of deep learning approaches on GPU (2-30 minutes) still takes relatively longer than the model training time of our approach on CPU (1-3 minutes).

(RQ4) How effective is our JITLine for prioritizing defective lines of a given defect-introducing commit?

Approach. To address this RQ, we first need to collect the line-level ground-truth data. To do so, we start from cloning a git repository of the studied projects. Then, we use PyDriller [91], a Python library for mining GitHub repository, to identify defect-fixing commits that are associated with each defect-introducing commit that is provided by McIntosh and Kamei [30]. Once identified, we examine the diff (a.k.a. code changes) made by the defect-fixing commits to identify lines that are modified/deleted by defect-fixing commits. Similar to prior work [55, 88], the lines that were modified or deleted by defect-fixing commits are identified as defective lines, otherwise clean. Then, we compare our JITLine with the state-of-the-art line-level JIT defect prediction approach by Yan *et al.* [23]. We implement the N-gram approach using the implementation provided by Hellendoorn *et al.* [75], Since Yan *et al.* [23] found that the Jelinek-Mercer (JM) smoothing method is the best choice, and the N-gram length has no substantial impact on the average performance, we followed their advice by using the Jelinek-Mercer (JM) smoothing method and the N-gram length of 6. Finally, we evaluate our approach and Yan *et al.* [23] using the following evaluation measures at the line level [23, 45]:

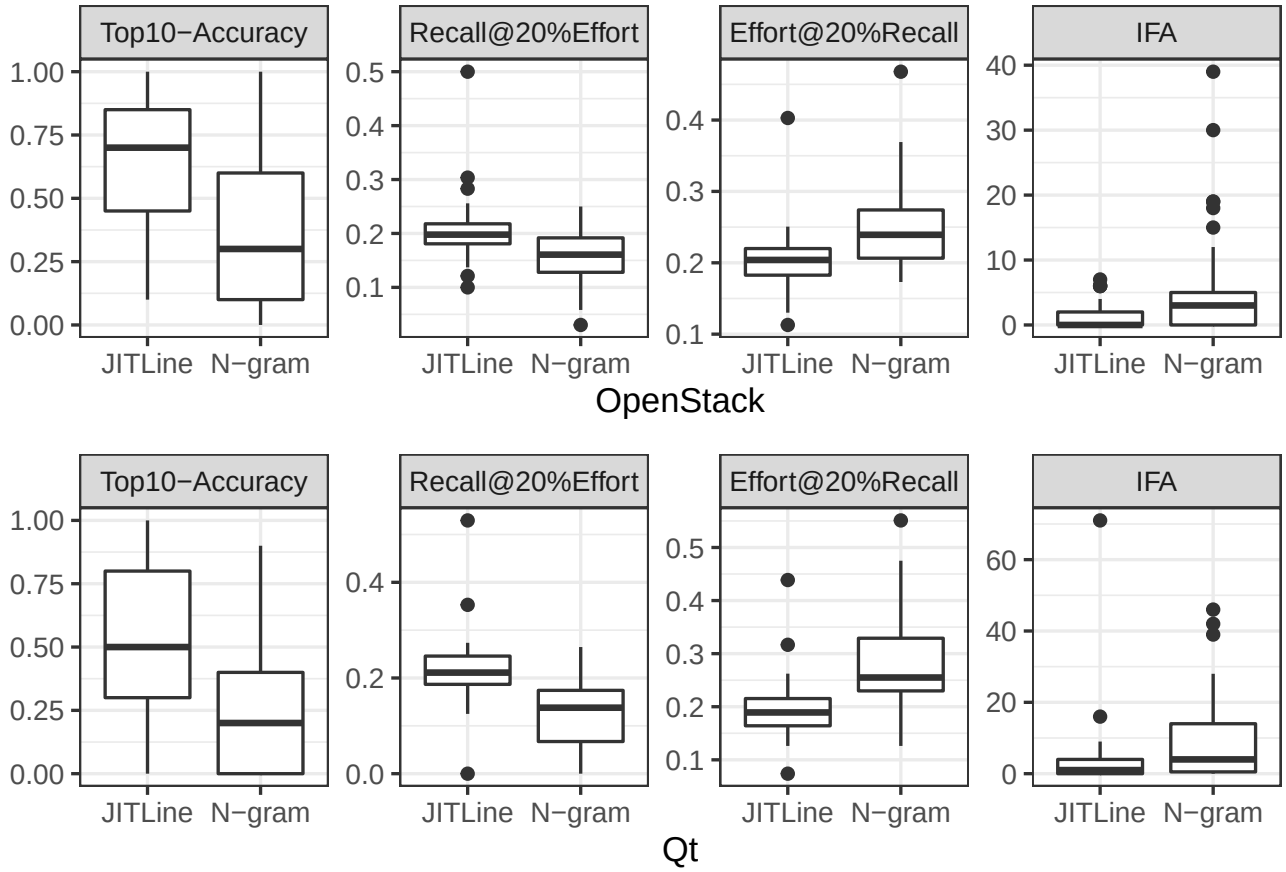


Figure 16: (RQ4) The results of our JITLine at the line level when compared to the N-gram-based line-level JIT defect prediction approach of Yan *et al.* [23] with respect to Top-10 Accuracy(↗), Recall@20%Effort(↗), Effort@20%Recall(↘), and IFA(↘). The higher (↗) or the lower (↘) the values are, the better the approach is.

1. Top-10 Accuracy measures the proportion of actual defective lines that are ranked in the top-10 ranking. Traditionally, developers may need to inspect all changed lines for a given commit—which is not ideal when SQA resources are limited. A high top-10 accuracy indicates that many of the defective lines are ranked at the top, which is considered effective.

2. Recall@20%LOC measures the proportion of defective lines that can be found (i.e., correctly predicted) given a fixed amount of effort (i.e., the top 20% of changed lines of a given defect-introducing commit). A high value of Recall@20%LOC indicates that an approach can rank many actual defective lines at the top.

3. Effort@20%Recall$_{\text{line}}$ measures the percentage of the amount of effort that developers have to spend to find the actual 20% defective lines of a given defect-introducing commit. A low value of Effort@20%Recall$_{\text{line}}$ indicates that the developers will spend a little amount of effort to find the 20% actual

defective lines.

4. Initial False Alarm (IFA) measures the number of clean lines that developers need to inspect until finding the first actual defective line for a given commit. A low IFA value indicates that developers only spend time inspecting only a few number of clean lines to find the first actual defective line.

Results. **Our JITLine approach is 133%-150% more accurate than the baseline approach by Yan *et al.* [23] for identifying actual defective lines in the top-10 recommendations.** Figure 16 shows that our approach achieves a median Top-10 Accuracy of 0.7 for OpenStack and 0.5 for Qt, while the baseline approach achieves a Top-10 Accuracy of 0.3 for Openstack and 0.2 for Qt. In addition, we find that our JITLine approach can find actual defective lines 25%-50% higher than the baseline approach, given the same amount of effort at 20%LOC. Figure 16 shows that our approach achieves a median Recall@20%LOC of 0.20 for OpenStack and 0.21 for Qt, while the baseline approach achieves a median Recall@20%LOC of 0.16 for OpenStack and 0.14 for Qt.

Our Wilcoxon signed-ranked test also confirms that the difference of Top-10 Accuracy and Recall@20%Effort between our approach and the baseline is statistically significantly (p-value < 0.05) with a Cliff's $|\delta|$ effect size of large ($|\delta| = 0.49 - 0.67$) for both Top-10 Accuracy and Recall@20%LOC.

Our JITLine approach requires 17%-27% less amount of effort than the baseline approach in order to find the same amount of actual defective lines. Figure 16 shows that our approach achieves a median Effort@20%Recall$_{\text{line}}$ of 0.20 for Openstack and 0.19 for Qt, while the baseline approach achieves a median Effort@20%Recall$_{\text{line}}$ of 0.24 for OpenStack and 0.26 for Qt. Similarly, our approach achieves a median IFA of 0 for OpenStack and 1 for Qt, while the baseline approach achieves a median IFA of 3 for OpenStack and 4 for Qt. Our Wilcoxon signed-ranked test also confirms that the difference of Effort@20%Recall$_{\text{line}}$ and IFA between our approach and the baseline is statistically significant (p-value < 0.05) with a Cliff's $|\delta|$ effect size of large ($|\delta| = 0.52 - 0.69$) for Effort@20%Recall$_{\text{line}}$ and a Cliff's $|\delta|$ effect size of medium ($|\delta| = 0.36 - 0.39$) for IFA.

3.7 Discussion

3.7.1 Implications to Practitioners

Our JITLine approach may help practitioners to better prioritize defect-introducing commits and better identify defective lines, since we find that our JITLine approach outperforms (RQ1), more cost-effective (RQ2), faster (RQ3), and more fine-grained (RQ4) than the state-of-the-art approaches (i.e., EALR, CC2Vec, and DeepJIT). Traditionally, Just-In-Time defect prediction methods only prioritize defect-introducing commits, saving a lot of code inspection effort. However, we find that the average ratio of actual defective lines for each commit is 50%. Thus, developers still spend unnecessarily effort on inspecting clean lines. In addition to predict defect-introducing commits, our JITLine approach can also accurately predict defective lines within a defect-introducing commit, saving 17%-20% effort that developers need to spend when compared to the baseline approach [23].

3.7.2 Implications to Researchers

Researchers should consider the key principles of Just-In-Time defect prediction models (i.e., to generate predictions as soon as possible), since the results of our replication study show that, when excluding testing datasets, the F-measure of CC2Vec approach is decreased by 38.5% for OpenStack and 45.7% for Qt. In reality, it is unlikely that the unlabelled testing dataset would be available beforehand when training JIT models. Thus, when conducting an experiment, testing data should be excluded when developing AI/ML models.

Researchers should explore simple solutions (i.e., Explainable AI approaches [33, 34, 45, 50, 92]) first over complex and compute-intensive deep learning approaches for SE tasks, since we find that our JITLine approach outperforms the deep learning approaches for Just-In-Time defect prediction. This recommendation has been advocated by prior studies in other SE tasks [75–78]. For example, Menzies *et al.* [78] suggested that researchers should explore simple and fast approaches before applying deep learning approaches on SE tasks. Hellendoorn [75] found that a careful implementation of NLP approaches outperform deep learning approaches. Liu *et al.* [77] found that simple k-nearest neighbours approach outperforms neural machine translation approaches.

3.7.3 Threats to Validity

Threats to construct validity relates to the impact of parameter settings of the techniques that our approach relies upon (i.e, SMOTE, DE, Random Forest, and LIME) [63, 72, 84]. To mitigate this threat, we apply a Differential Evolution algorithm to optimize the parameter setting of the SMOTE technique. We use the parameter settings of DE, suggested by Agrawal *et al.* [62]. We use the default settings of LIME (i.e., the number of samples = 5,000). For the baseline approaches, we use the best parameter settings provided by the implementation of the DeepJIT [11] and CC2Vec approaches [12].

Prior work raised concerns that the ground-truths data collection of defect-introducing commits could be delayed [93, 94]. Thus, it is possible that our studied JIT datasets may be missing some of the false negative commits when defects are not fixed (i.e., defect-fixing commits that are not yet fixed). However, the goal of this paper is not to improve the data construction approach. Instead, we use the same datasets that were used in the prior work for a fair comparison. Thus, future work should consider addressing this concern.

Threats to external validity relates to the limited number of the studied datasets (i.e., OpenStack and Qt) to ensure a fair comparison with the CC2Vec approach [12]. Thus, other commit-level datasets can be explored in future work.

Threats to internal validity relates to the randomization of several techniques that our approach relies upon [95]. After we repeat our experiments with different random seeds, we observe minor differences (e.g., ± 0.01 for AUC). Nevertheless, our JITLine approach is still the best performer for all RQs. The used random seed number is reported in our replication package at Zenodo: `http://doi.org/10.5281/zenodo.4433498`.

We follow the experimental setting of the original study [11, 12] (i.e., one single training/testing data split without cross-validation). Therefore, statistical analysis and effect size analysis are not applied for RQ1, RQ2, and RQ3, since we have only one performance value for each project.

3.8 Summary

In this chapter, we propose JITLine approach, a machine learning-based JIT defect approach for predicting defect-introducing commits and identifying defective

lines that are associated with that commit. Then, we conduct our empirical study to demonstrate that our JITLine approach is better (RQ1), more cost-effective (RQ2), faster (RQ3) and more fine-grained (RQ4) than the state-of-the-art JIT defect prediction approaches (i.e., EARL, DeepJIT, and CC2Vec).

Therefore, our JITLine approach may help practitioners to better prioritize defect-introducing commits and better identify defective lines. In addition, our results highlight the negative impact of excluding testing datasets in model training and the importance of exploring simple solutions (e.g., explainable AI approaches) first over complex and compute-intensive deep learning approaches.

4 DeepLineDP: Towards a Deep Learning Approach for Line-Level Defect Prediction

Chapter Overview

Deep learning-based software defect prediction models have been proposed to help practitioners prioritize SQA resources on the files that are most likely to introduce defects after software is released. However, such models in prior works lack contextual information. Thus, they still cannot locate defective lines in the predicted defective files.

In this chapter, to address the overarching research question of the thesis, we propose DeepLine that leverages code tokens from surrounding lines in a source code file to predict defective files and locate defective lines. Through a case study of 32 releases of 9 software projects, we find that DeepLineDP is more accurate than other file-level defect prediction approaches and is more cost-effective than other line-level defect prediction approaches.

The work in this chapter appears in Chanathip Pornprasit, and Chakkrit Tantithamthavorn. "Deeplinedp: Towards a deep learning approach for line-level defect prediction." In IEEE Transactions on Software Engineering (TSE) 49.1 (2022), pp. 84–98.

4.1 Introduction

Software defects are prevalent and costly. Thus, Software Quality Assurance (SQA) practices play a critical role to ensure the absence of software defects. Despite several SQA tools having been heavily invested during the development phase (e.g., CI/CD, code review, static analysis), software defects may still slip through to the official releases of a software product (i.e., post-release defects) [4, 40]. Vassallo *et al.* [96] found that practitioners still control for quality only at the end of a sprint and at the release preparation stage. However, real-world software projects are extremely large and complex. Hence, it is intuitively infeasible to exhaustively perform SQA activities for all of the files of the codebase.

To address this challenge, defect prediction is proposed to help developers prioritize their limited SQA resources on the most risky files that are likely to have post-release software defects. Recent studies found that deep learning approaches, which automatically learn syntactic and semantic features, outperform traditional machine learning approaches that used process and product as features metrics [97–99]. However, there exist two main limitations in prior studies.

(1) The granularity levels of deep learning-based defect predictions are still coarse-grained. Defect prediction models have been proposed at various levels of granularity (e.g., packages [74], components [40], modules [100], files [74, 79, 82, 97, 98], methods [101], and commits [11, 12, 25]). However, not all lines in a defective file are actually defective. Therefore, developers still waste a large amount of SQA effort to inspect clean lines that may not lead to post-release defects (i.e., Table 6 shows that only 0.03%-2.9% lines of the whole release are defective). As such, line-level defect prediction is needed to help developers prioritize their SQA effort on the high-risk areas of source code so SQA effort can be allocated in a cost-effective manner. However, line-level defect prediction is a challenging problem and still remains largely unexplored (e.g., no existing deep learning-based approach considers sequence of tokens).

(2) The surrounding tokens and the surrounding lines have not yet been fully utilized. Source code has a hierarchical structure (i.e., tokens forming lines and lines forming files) and is contextually dependent. Thus, the same code token that appears in different lines may have different lexical meaning depending on its location (e.g., variable declaration or assigning a value to a variable). Therefore, the riskiness of code tokens should be different depending on their location. However, current deep learning approaches for file-level defect pre-

diction [97–99, 102] can capture only the long-term sequences of code tokens without considering surrounding lines, assuming that the same code token at different lines is equally important to predict defective files—which is likely not true.

In this paper, we began with conducting a survey study to investigate the state-of-practice of code inspection in modern code review, and practitioners' perception on a line-level defect prediction tool. We recruited participants through social media platforms targeting professionals and software developers (i.e., LinkedIn, Facebook Groups). Through an analysis of the 36 responses obtained from the participants, we addressed the following two motivating questions (MQs):

(MQ1) How do practitioners perform code inspection in modern code review process?
Results. Given a changed file in a pull request, 80.6% of the respondents currently inspect source code in a top-down order. Once defective lines are identified, 72.2% of the respondents inspect the defective lines and their related method calls, while 52.8% of the respondents inspect the defective lines and their surrounding lines. Also, 50% of the respondents spent at least 10 minutes to more than one hour to review a single file, indicating that current code review activities are still time-consuming. Importantly, 64% of the respondents perceived that code inspection activity is very challenging to extremely challenging.

(MQ2) Would a line-level defect prediction tool be helpful for practitioners?
Results. 44% of the respondents perceived that a line-level defect prediction tool would potentially be helpful in identifying defective lines. In addition, 64% of the respondents would consider using the tool if it is publicly available for free.

Motivated by the findings from our survey with practitioners, we present DeepLineDP, a deep learning approach to learn the surrounding code tokens and the surrounding lines of source code in order to predict defective files and locate defective lines, which performs as follows. First, we generate a vector representation for each code token. Then, we employ a bidirectional GRU unit [103] to capture the surrounding tokens of that source code line. Next, we employ a Hierarchical Attention Network (HAN) architecture [104] to learn the hierarchical structure of source code to predict defective files. After that, we compute the risk score of code tokens that contribute to the prediction of a given defective file using the

attention mechanism [105]. We then generate a ranking of risky lines using the summation of the risk score of the code tokens that appear in that line. Through a case study of 32 releases of 9 software projects, we address the following four research questions (RQs):

(RQ1) Can our DeepLineDP be used to differentiate the riskiness of code tokens in defective and clean lines?
Results. The risk score of the same code token that appears in different lines varies greatly depending on their locations, since we observe a maximum difference of 1. For example, the risk score of a code token may be extremely risky in one line (a risk score of 1), while not being risky at all in another line (a risk score of 0). In addition, we find that the risk score of code tokens in defective lines are significantly higher than the risk score of code tokens in clean lines, suggesting that our DeepLineDP approach can be used to differentiate the riskiness of code tokens in defective and clean lines.

(RQ2) What is the accuracy of our DeepLineDP for predicting defective files?
Results. Our DeepLineDP achieves a median AUC of 0.81 and a median Balanced Accuracy of 0.63, which is 17%-37% and 3%-26% more accurate than the state-of-the-art in terms of the median AUC and the median Balanced Accuracy, respectively. The ScottKnott ESD test also confirms that our DeepLineDP approach always appears at the top rank for both measures.

(RQ3) What is the cost-effectiveness of our DeepLineDP for locating defective lines?
Results. Our DeepLineDP is 47%-250% more cost-effective than the state-of-the-art line-level defect prediction approaches for Recall@Top20%LOC, while achieving 38% and 49% less Effort@Top20%Recall than using the ErrorProne static analysis tool and the N-gram model, respectively.

(RQ4) What is the accuracy of our DeepLineDP for line-level cross-project defect predictions?
Results. Our DeepLineDP models are transferable to other software projects, since our DeepLineDP models still achieve an AUC of 0.63-0.79 and a Recall@Top20%LOC of 0.31-0.46. However, they can be slightly less accurate and less cost-effective than the models that are trained from the previous release of its own project. Nevertheless, our DeepLineDP models still achieve a reasonable performance.

Novelty and Contributions. To the best of our knowledge, the novelty and main contributions of our work in this chapter are as follows:

1. A survey with 36 practitioners to understand their current code inspection practices in modern code review and their perceptions to adopt a line-level defect prediction.
2. DeepLineDP—a deep learning-based approach for line-level defect prediction that aims to automatically learn the surrounding tokens and surrounding lines to predict defective files and defective lines.
3. An extensive experiment using both within-project and cross-project evaluation settings at the file and line levels with 3 traditional measures and 3 effort-aware measures.
4. Our empirical finding shows that our DeepLineDP models (1) can differentiate the riskiness of the same code token that appears in different lines; (2) can accurately predict the defective files; (3) can effectively generate the ranking of defective lines; and (4) trained on a software project can be transferable to other software projects while achieving a reasonable performance.

To foster the replication of our study, we publish the implementation of our DeepLineDP and the baselines at https://github.com/awsm-research/DeepLineDP.

Paper Organization. The rest of this paper is organized as follows. Section 4.2 presents a survey study of the current code inspection practices in modern code review. Section 4.3 presents the architecture of our DeepLineDP approach. Section 4.4 presents the experimental design and results. Section 4.5 discloses the threats to the validity. Section 4.6 discusses the related work. Finally, we draw conclusions in Section 4.7.

4.2 A Motivating Survey

In this section, we aim to investigate (1) how practitioners perform code inspection in modern code review and (2) would a line-level defect prediction tool be helpful for practitioners. Below, we described the approach to conduct a survey and presented the results from our survey study.

4.2.1 Approach

Similar to Kitchenham and Pfleeger [106], we conducted our study according to the following steps: (1) design and develop a survey, (2) recruit and select participants, and (3) verify data and analyze data. We explained the detail of each step below.

(Step 1) Design and develop a survey. The purpose of our survey is to investigate the current practices in modern code review process. We designed our survey as a cross-sectional study where participants provided their responses at one fixed point in time. The survey consists of 7 closed-ended questions and 6 open-ended questions. For closed-ended questions, we used multiple-choices question and Likert scale from 1 to 5. Our survey consists of three parts: preliminary question; understanding current code inspection practices; and practitioners' perceptions on a line-level defect prediction tool. Our preliminary question starts with (*"Do you perform code review"*) to ensure that our survey results obtained from the right target participants, followed by roles, levels of experience, and primary programming language skills. The next part is focused on a set of questions to understand the current code inspection practices. Then, the final part is focused on practitioners' perceptions on a line-level defect prediction tool. To do so, we presented a usage scenario and an example visualization. Figure 17 is an example defective file that is correctly predicted by our DeepLineDP. The shade colour of each code token varies based on the risk score from dark red (very risky) to light red (less risky). The example was obtained from the file `../store/Directory.java` from the release 3.1 of the Lucene project, which has 426 lines of code. With our DeepLineDP, the model correctly predicts that the line number of 227-228 are the most risky lines with a risk score of 0.99 and 0.97, respectively due to the tokens `createOutput()` and `openInput()`. We used Google Form to create our online survey. When accessing the survey, each participant was provided with an explanatory statement that describes the purpose of the study, why the participant is chosen for this study, possible benefits and risks, and confidentiality. The survey takes approximately 10 minutes to complete and is anonymous. Our survey has been rigorously reviewed and approved by the Monash University Human Research Ethics Committee (MUHREC ID: 30739).

(Step 2) Recruit and select participants. We recruited the practitioners via LinkedIn since we can specify the target group that we would like to access. We then sent a survey invitation to the target groups via the LinkedIn direct message. To ensure that our survey is not biased, we selected participants from various

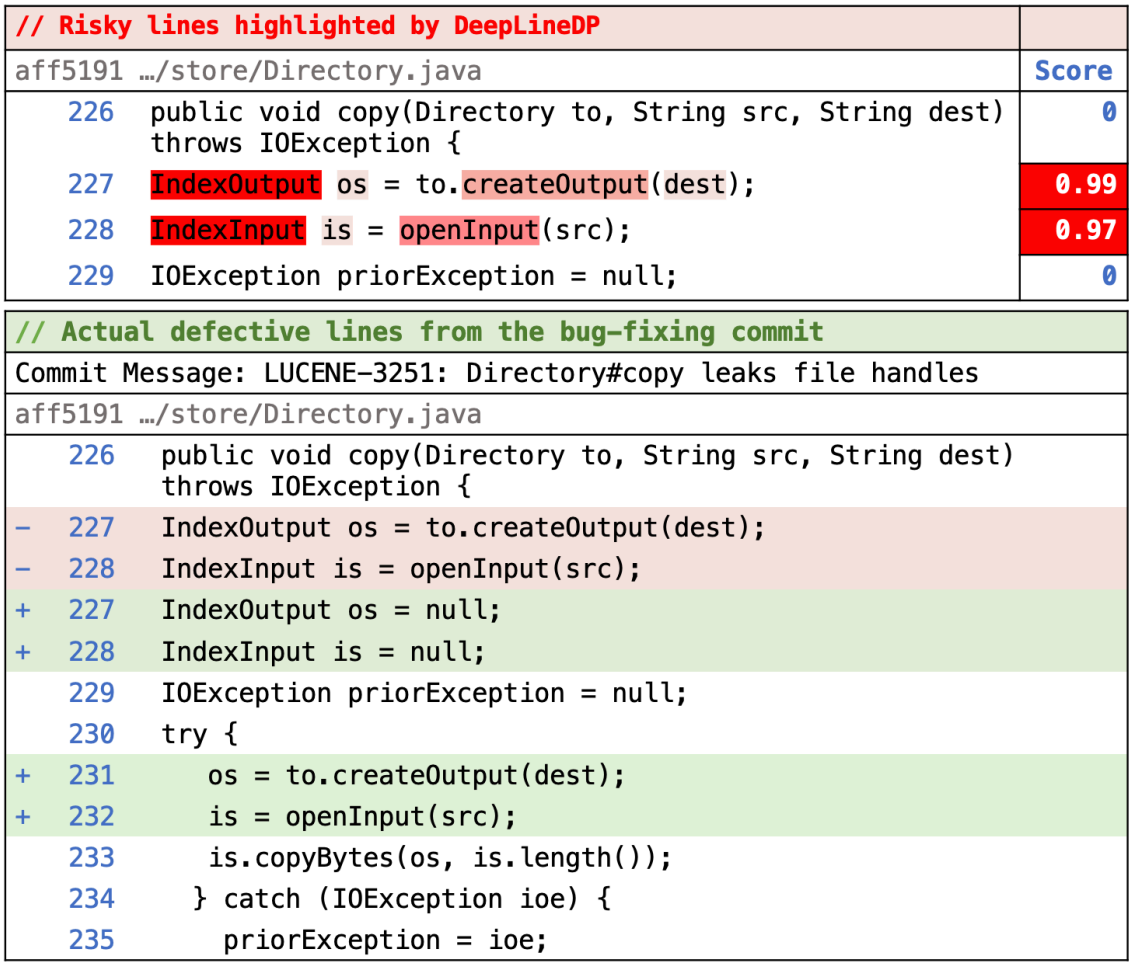


Figure 17: An example visualization to highlight the most risky tokens obtained from our DeepLineDP approach.

large companies. Due to the limitation of the number of invitations in LinkedIn, we were able to sent at most 100 invitations per week. In total, we sent a survey invitation to 100 practitioners via LinkedIn. We received a response rate of 15% ($\frac{15}{100}$). Then, we also received 28 additional responses through the survey advertisement at several software developer's communities (e.g., Facebook groups). Finally, we obtained a total of 43 responses over one week recruitment.

(Step 3) Verify and analyze data. To verify the completeness of the response in our survey (i.e., whether all questions were appropriately answered), we manually read all of the open-ended questions. to ensure that our survey results are derived from the right target participants, we excluded 7 responses that do not perform code review. Finally, we obtained a set of 36 responses. We presented the results of closed-ended responses in a Likert scale with stacked bar plots. We manually analyzed the responses of the open-ended questions to extract in-depth insights.

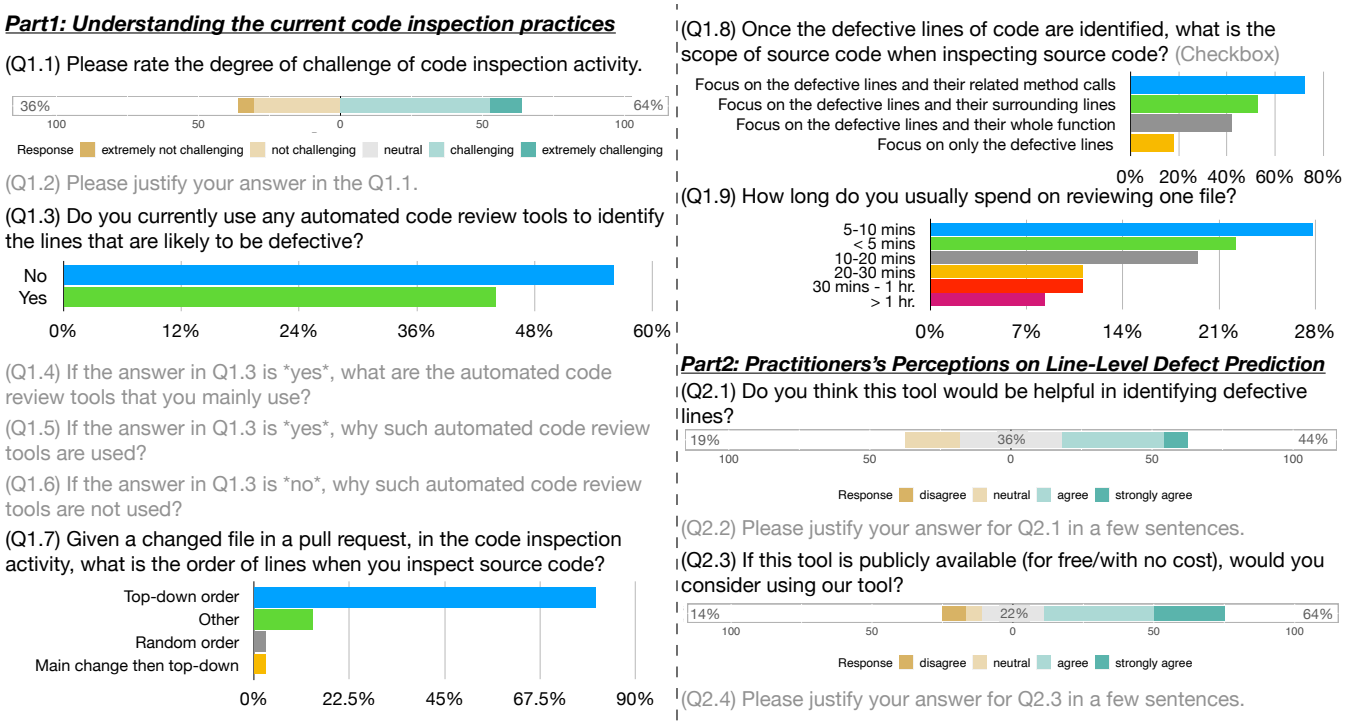


Figure 18: (MQ1/MQ2) A summary of the survey questions and the results obtained from 36 participants.

4.2.2 Respondent Demographics

Among all of the 36 respondents in the survey, they have the following roles in a software development team: (full-stack) software engineer or software developer (88.8%), quality assurance engineer (5.6%) , team leader (2.8%), others (2.8%). The respondents have different Years of Professional Experience: less than 5 years (52.8%), 6-10 years (33.3%), 11-15 years (8.3%), 16-20 years (2.8%), more than 20 years (2.8%) The respondents described their experience in programming languages as Java (24.8%), Javascript (16.7%), C# (16.7%), Python (13.9%), C/C++ (5.6%), Go (8.3%), Kotlin (5.6%), PHP (2.8%), Scalar (2.8%), Swift (2.8%). These demographics indicate that the responses are collected from practitioners who have a variety of roles, years of experiences, and various programming language expertise. Thus, our findings are likely not bound to specific groups of practitioners.

4.2.3 Survey Results

Figure 18 shows a summary of the survey questions and the results of MQ1 and MQ2.

(MQ1) How do practitioners perform code inspection in modern code review process?

Findings. **Developers inspect source code in a top-down order. Once defective lines are identified, they will inspect the method calls and their surrounding lines. Each file often takes at least 10 minutes to more than one hour.** To speed up code review process, automated code review tools are currently used. However, 64% of the respondents still perceived that their current code inspection activity remains very challenging, highlighting the need of a line-level defect prediction tool to help developers identify the areas of code that are likely to be defective in the future. Below, we summarize the key findings from our survey.

- Given a changed file in a pull request, 80.6% of the respondents currently inspect source code in a top-down order (i.e., from the top to the bottom). Other respondents inspect source code in a random order, or inspect the main code change followed by the top-down order.
- Once the defective lines are identified, 72.2% of the respondents focused on inspecting the defective lines and their related method calls, while 52.8% of the respondents inspect the defective lines and their surrounding lines, 41.7% of the respondents inspect the defective lines and their whole function, and 17.6% of the respondents inspect only the defective lines.
- 50% of the respondents spent at least 10 minutes to more than one hour to review a single file, indicating that current code review activities are still time-consuming.
- To aid code review process, 44.4% of the respondents currently use automated code review tools (e.g., SonarQube and ESLint), since such tools can facilitate their code review tasks (R10: *It provides massive rules for code reviews, support many languages and the review report is intelligible*, R25: *Help us highlight and simplify the work that they need to do.*)
- 64% of the respondents perceived that code inspection activity is very challenging to extremely challenging. They stated the reasons that it's challenging to review code that they do not own (R20: *It's challenging when I have to review the code in the repo that I do not own*), understanding the logic behind code is time consuming (R24: *People have different way of thinking. My time is usually spent on understanding their reasoning behind the code before I leave a comment to challenge their design or logic.*), and there are a lot of code to review (R26: *Many lines of code to be reviewed*).

(MQ2) Would a line-level defect prediction tool be helpful for practitioners?

Findings. **44% of the respondents perceived that a line-level defect prediction tool would potentially be helpful in identifying defective lines**, due to various reasons:

- Reduce time to do code review (R18: *It could be useful and take less time to review the code*).
- Easy to understand (R8: *the highlight and score are clear and easy to understand*, R26: *Red alert attracts your eyes so it is easier to spot what is wrong*).
- Provide some useful information (R14: *Show highlight with probability is informative*).

In addition, we find that the respondents who are (full-stack) software engineers or software developers, have less than 5 years of experience, and focus on the surrounding lines and the whole function of the identified defective lines, tend to perceive that the tool is helpful in identifying defective lines.

64% of the respondents would consider using a line-level defect prediction tool if it is publicly available for free, due to various reasons:

- Expedite software development process (R12: *If it make process go faster, why not?*).
- Interesting (R14: *It's interesting to use the tools*).

4.3 Learning the Surrounding Tokens and Surrounding Lines for Line-Level Defect Prediction

In this section, we present DeepLineDP, an approach to address the challenges of line-level defect prediction. Our DeepLineDP approach is designed to capture defective code lines through the use of bidirectional GRU unit to learn the surrounding tokens and surrounding lines using a hierarchical attention network. Figure 19 provides an overview of our approach.

Overview. Our approach begins with data collection and data preparation to generate datasets that contains source code files, file-level and line-level ground-truth labels (i.e., files that are affected by a post-release defect and lines that

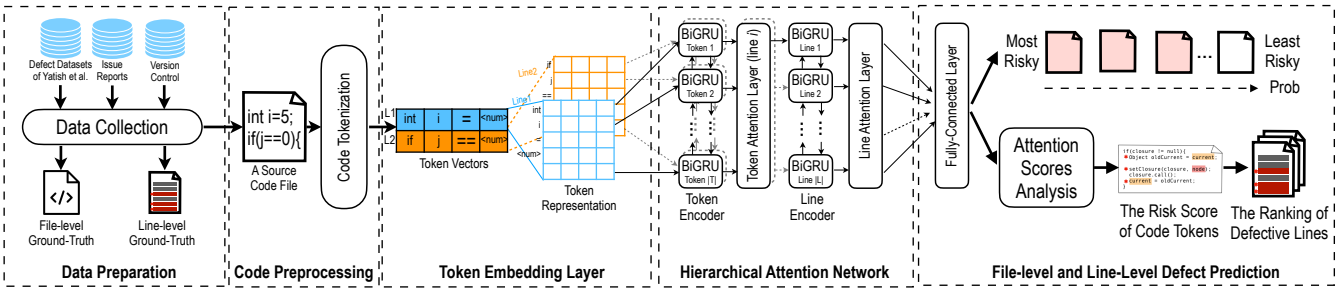


Figure 19: An overview diagram of our DeepLineDP approach.

changed or deleted to address the defect). For each source code file, we perform several preproceessing steps, including code abstraction and vocabulary size management. Next, we perform source code representation to generate a vector representation. After that, we use a hierarchical attention network to learn the hierarchical structure of source code for file-level defect prediction, while enabling line-level defect prediction. Then, the prediction layer produces the probability score of the prediction of defective files. Finally, we use the token-level attention layer to identify defective lines based on the most important tokens that contribute to the prediction of defective files.

4.3.1 Source Code Preprocessing

The goal of the source code preprocessing steps is to extract syntactic, semantic, and contextual code features of each code token. For each Java file, we use JavaParser [3] to build an Abstract Syntax Tree (AST) to generate a stream of tokens and determine their type by extracting syntactic information (e.g., whether a token represents an identifier, a method declaration). Then, each source code file is parsed into a set of lines and each line is parsed into a sequence of code tokens.

Vocabulary Size Management. A large vocabulary size often requires high computing time and large memory usage [13, 97, 107, 108]. To alleviate this issue, we replace integers, real numbers, exponential notation, and hexadecimal numbers with a generic ⟨*num*⟩ token, and replace constant strings with a generic ⟨*str*⟩ token. We remove special characters (i.e., {. , ” ’ ( ) : (space) } !) since Rahman and Rigby [83] found that these special tokens introduce noise to prediction models. We replace tokens which exist in test sets but do not exist in the training set with a special token ⟨*unk*⟩, known as the out-of-vocabulary problem. We ignore

[3] http://javaparser.org/

blank lines, as they do not contribute to the actual behavior of the code.

4.3.2 Token Embedding Layer

Unlike prior studies that represent source code as a long-term sequence [97], our goal is to maintain the hierarchical structure of source code (i.e., tokens forming lines and lines forming files). Thus, we represent the source code file by maintaining the order of lines and tokens in the original code. Each file is now a sequence of lines $\langle l_1, l_2, ..., l_n \rangle$ and each line is a sequence of code tokens $\langle w_1, w_2, ..., w_n \rangle$. For each token, we generate a vector representation using the Word2Vec function provided by the gensim Python library. In particular, we build a project-specific language model using our training dataset for each project to ensure that the vector representation is derived from the domain-specific vocabularies and to obtain the optimal distributed representations. That means the vector representations generated from domain-specific language models tend to produce more meaningful vectors (i.e., capturing better distributed relationships between tokens and their surrounding tokens better than a generic pre-trained language model). When training the Word2Vec model, we use the Continuous Bag of Words (CBOW) architecture to learn the distributed representations of tokens, since the CBOW architecture considers the surrounding tokens to generate a vector representation of the target token.

4.3.3 Learning the Hierarchical Structure of Source Code

We use Hierarchical Attention Network [104] to learn the hierarchical structure of source code (see Figure 20). Specifically, we use a two-layer attention network (a token layer and a line layer). This network consists of four parts: a token encoder, a token-level attention layer, a line encoder, and a line-level attention layer. Assuming that a source code file $f \in \mathcal{F}$ has a sequence of lines $\mathcal{L} = [l_1, l_2, ..., l_{|\mathcal{L}|}]$, where l_i contains a sequence of tokens $\mathcal{T} = [w_{i1}, w_{i2}, ..., w_{i|\mathcal{T}|}]$, where w_{it} represents the code token in the i-th line, $t \in [1, |\mathcal{T}|]$.

Token Encoder. Given a line l_i with a sequence of tokens $\mathcal{T}$ and a word embedding matrix $\mathbf{W} \in \mathbb{R}^{|\mathcal{V}^{\mathcal{C}}| \times d}$, where $\mathcal{V}^{\mathcal{C}}$ is the vocabulary containing all tokens extracted from training source code files and d is the word embedding size of the representation of tokens, we first obtain a vector representation of each token for each line $v_{it} = \mathbf{W}(w_{it})$ from Word2Vec, where v_{it} indicates the vector representation of token w_{it} in the word embedding matrix $\mathbf{W}$.

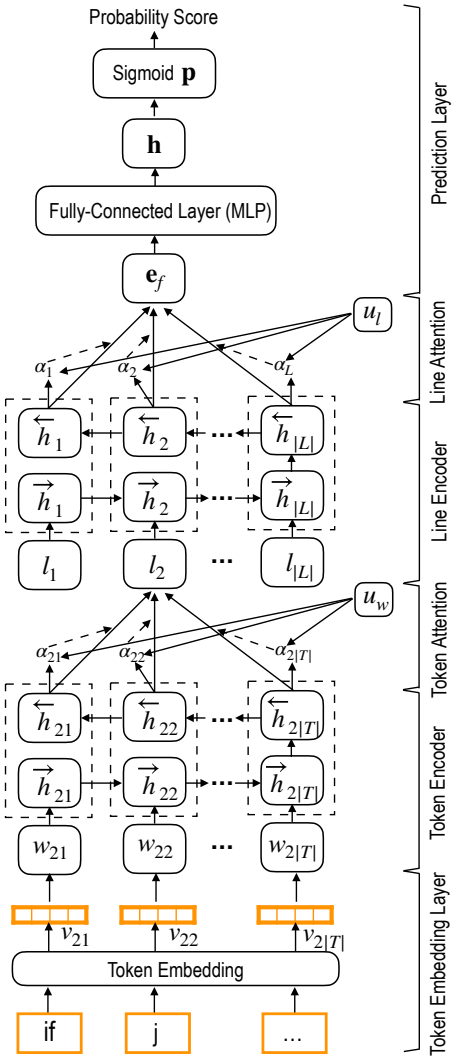


Figure 20: An overview of a two-layer hierarchical attention network to learn the hierarchical structure of source code using a bidirectional GRU unit to capture the code context (i.e., surrounding tokens).

We employ a bidirectional GRU [103] to summarize information from the context of a token in both directions. GRU is proposed to solve the vanishing gradient problem, which is commonly found in a standard RNN architecture when training with a long sequence of tokens. To capture the contextual information, the bidirectional GRU includes a forward GRU $\overrightarrow{h}_{it} = \overrightarrow{\text{GRU}}(v_{it}), t \in [1, |\mathcal{T}|]$ that reads the line l_i from left (w_{i1}) to right ($w_{i|T|}$) and a backward GRU $\overleftarrow{h}_{it} = \overleftarrow{\text{GRU}}(v_{it}), t \in [|\mathcal{T}|, 1]$ that reads the line l_i from right ($w_{i|T|}$) to left (w_{i1}). We obtain an annotation for a given token w_{it} by concatenating the forward hidden state $\overrightarrow{h}_{it}$ and backward hidden state $\overleftarrow{h}_{it}$, i.e., $h_{it} = [\overrightarrow{h}_{it} \oplus \overleftarrow{h}_{it}]$.

Token Attention. Based on the intuition that not all tokens contribute equally to

the semantic representation of the source code line, we use the attention mechanism [105] to highlight the tokens that are more important to the semantics of the source code lines and aggregate the representation of those informative tokens to form a line vector.

We first feed the token annotation h_{it} through a one-layer Multi-Layer Perceptron (MLP) to get a hidden representation (u_{it}) of h_{it}, i.e., $u_{it} = \tanh(\mathbf{W}_w h_{it} + b_w)$. Similar to prior studies [12, 104], we define a token-level context vector (u_w) that can be seen as a high level representation of the answer to the fixed query "what is the most informative token" over the tokens. The token context vector u_w is randomly initialized and learned during the training process. Then, we measure the importance of the token as the similarity of u_{it} with a token-level context vector u_w. Then, we compute a normalized contribution (attention) α_{it} of token w_{it} of the line l_i through a softmax function [109]: $\alpha_{it} = \frac{exp(u_{it}^T u_w)}{\sum_t exp(u_{it}^T u_w)}$. For each line l_i, its vector is computed as a weighted sum of the embedding vectors of the tokens based on their importance as follows: $s_{it} = \sum_t \alpha_{it} h_{it}$.

Line Encoder. Given source code lines (i.e., l_i), we use a bidirectional GRU to encode the lines as follows: $\overrightarrow{h}_i = \overrightarrow{\text{GRU}}(l_i), i \in [1, |\mathcal{L}|]$ and $\overleftarrow{h}_i = \overleftarrow{\text{GRU}}(l_i), t \in [|\mathcal{L}|, 1]$. Similar to the Token Encoder, we obtain an annotation of the source code line l_i by concatenating the forward hidden state $\overrightarrow{h}_i$ and the backward hidden state $\overleftarrow{h}_i$ of this line. The annotation of the line l_i is denoted as $h_i = [\overrightarrow{h}_i \oplus \overleftarrow{h}_i]$, which summarizes the line l_i considering its neighboring lines.

Line Attention. We again use an attention mechanism to highlight the lines that are more important to the semantics of the source code files and aggregate the representation of those informative lines to form a file vector. We first define a line-level context vector $u_l = \tanh(\mathbf{W}_l h_i + b_s)$. Then, we compute a normalized contribution (attention) α_i of line u_l of the line l_i through a softmax function [109]: $\alpha_i = \frac{exp(u_l^T u_s)}{\Sigma_i exp(u_l^T u_s)}$. Here u_s is a sentence-level context vector which is compared with u_l to measure the importance of a code line. For each file, $\mathbf{e}_f$ is the embedding vector of the file that is computed as a weighted sum of the embedding vectors of the lines based on their importance as follows: $\mathbf{e}_f = \Sigma_i \alpha_i h_i$.

4.3.4 File-level and Line-Level Defect Prediction Layer

The embedding vector $\mathbf{e}_f$ is a high level representation of the source code file which can be used as features for file-level defect prediction. This vector is fed

to a one-layer Multi-Layer Perceptron (MLP) (a.k.a. a fully-connected layer) to produce a prediction score $\mathbf{h} = \mathbf{w}_h \cdot \mathbf{e}_f + b_h$, where $\mathbf{w}_h$ is the weight matrix used to connect the embedding vector $\mathbf{e}_f$ with the hidden layer, and $\mathbf{b}_h$ is the bias value. Finally, the prediction score is passed to an output layer to compute a probability score for a given source code file. We use Sigmoid function to compute the predicted probability of a given source code file as follows: $\mathbf{p}(y_i = 1|f_i) = \frac{1}{1+\exp(-\mathbf{h}\cdot\mathbf{w}_0)}$, where $y_i \in \mathbf{Y}$ is the probability score of the i-th file and f_i is the file that we want to predict.

To identify defective lines, we first extract the attention score, ranges from -1 to 1, of each code token in that defective file. The attention score, calculated by the attention mechanism [105], is used as a proxy to indicate the riskiness of code tokens. The concept of the attention score is similar to the importance score of the random forest's variable importance that is widely-used in software engineering, but having different calculations. To do so, we rank the attention score obtained from the token attention layer in a descending order. Then, we select the top-k tokens that have the highest attention scores. Then, we compute the line-level risk score as the summation of the risk score of any tokens that appear in the top-k. Finally, we produce a ranking of the most risky lines based on the risk scores of each line. In this paper, we choose the k of 1,500.

4.4 Study Design and Results

The goal of this paper is to empirically assess our hypothesis whether the hierarchical structure and code context (i.e. surrounding tokens and surrounding lines) should be considered to identify the fine-grained locations of defective code (i.e., defective files and defective lines). Below, we present the study design followed by the results.

Line-level Defect Datasets. In this paper, we use the benchmark line-level defect dataset prepared by Wattanakriengkrai *et al.* [45]. The dataset consists of 32 software releases that span across 9 open-source software systems. Table 6 shows a statistical summary of our studied dataset. Each release contains 731 to 8,846 files, 74,349-567,804 lines of code, and 58,659-621,238 code tokens. Below, we discuss the detailed steps that were used to collect file-level and line-level ground-truths.

(Step-1) Collect issue reports. For each studied system, all issue reports were retrieved from the JIRA Issue Tracking System (ITS). Then, the unique identifier

of the issue reports (IssueID), the issue report type (e.g., a bug or a new feature), and the affected releases (i.e., the releases that are affected by a given issue report) were extracted. Since the goal of release-based defect prediction is to predict if a file will be affected by an issue report in the future, only the issue reports that were classified as bug, were reported after the studied release, and affected the studied releases were focused.

(Step-2) Collect a snapshot of source code. For each release, a snapshot of source code at the release date (i.e., the list of files of a given release) was collected from the Git Version Control System (VCS).

(Step-3) Identify defect-fixing commits. **Defective files** are defined as files that are affected by post-release defects [54, 110]. To identify defective files, commits that addressed a post-release defect (i.e., defect-fixing commits) were needed to be found. To do so, defect-fixing commits that are associated with the defect reports that affected the studied release were found, using regular expressions to search for the issue IDs in the commit messages. Then, files that are fixed for the defect reports were labelled as defective, otherwise clean.

(Step-4) Identify defective lines in the code snapshot at the release date. **Defective lines** are defined as lines in the code snapshot at the release date that are affected by a post-release defect report [111, 112]. To identify defective lines, the defective files (in the defect-fixing commits) were examined and identified which lines in the code snapshot at the release date are fixed or modified (i.e., modification or deletion operations) to address defects. Then, lines that are fixed or modified for the post-release defect reports were labelled as defective, otherwise clean.

With these four steps, line-level defect datasets, which is a snapshot of source code at the release date and defective files and defective lines at the release date that later are fixed or modified to address for software defects, could be produced.

Training Details. The implementation of our DeepLineDP model is based on Pytorch, an open-source deep learning framework. The experiment is run on a computer with an AMD Ryzen 9 5950X 16-Core @3.4 GHz, a RAM of 64GB, and an NVIDIA RTX 3090 GPU with 24 GB memory. The average model training time is 5 minutes. Below, we describe the hyperparameter settings that we used for our approach.

Table 6: An overview of the studied projects.

System	Description	#Files	#LOC	#Code Tokens	#Unique Tokens	%Defective Files	%Defective Lines	Studied Releases
ActiveMQ	Messaging and Integration Patterns	1,884-3,420	142k-299k	141k-293k	16,312	2%-7%	0.08%-0.44%	5.0.0, 5.1.0, 5.2.0, 5.3.0, 5.8.0
Camel	Enterprise Integration Framework	1,515-8,846	75k-485k	94k-621k	12,202	2%-8%	0.09%-0.24%	1.4.0, 2.9.0, 2.10.0, 2.11.0
Derby	Relational Database	1,963-2,705	412k-533k	251k-329k	63,677	6%-28%	0.10%-0.63%	10.2.1.6, 10.3.1.4, 10.5.1.1
Groovy	Java-syntax-compatible OOP	757-884	74k-93k	58k-68k	16,901	2%-4%	0.10%-0.17%	1.5.7, 1.6.0.Beta_1, 1.6.0.Beta_2
HBase	Distributed Scalable Data Store	1,059-1,834	246k-537k	149k-257k	30,155	7%-11%	0.17%-1.02%	0.94.0, 0.95.0, 0.95.2
Hive	Data Warehouse System for Hadoop	1,416-2,662	290k-567k	147k-301k	29,245	6%-19%	0.31%-2.90%	0.9.0, 0.10.0, 0.12.0
JRuby	Ruby Programming Lang for JVM	731-1,614	106k-240k	72k-165k	21,630	2%-13%	0.03%-0.09%	1.1, 1.4, 1.5, 1.7
Lucene	Text Search Engine Library	805-2,806	101k-342k	76k-282k	17,733	2%-8%	0.07%-0.39%	2.3.0, 2.9.0, 3.0.0, 3.1.0
Wicket	Web Application Framework	1,672-2,578	106k-165k	93k-147k	21,993	2%-16%	0.05%-0.46%	1.3.0.beta1, 1.3.0.beta2, 1.5.3

We use the word embedding vector size d of 50 to train Word2Vec model. We use the learning rate of 0.001. The model is trained at 10 epochs. The batch size is set to 32. We use the binary cross-entropy as the loss function. We use the Adam optimizer [113] to minimize the loss function, since it has been shown to be computationally efficient and require low memory consumption. We use Dropout (the dropout ratio is 0.5) and Layer Normalization to prevent overfitting. The reason for this is that Watson *et al.* [114] found that the Dropout is one of the most commonly-used techniques to prevent overfitting and underfitting in software engineering tasks. We do not employ the early stopping technique, since we found that the characteristics of our release-based datasets are different from release to release. Thus, the problem of release-based defect prediction is different from other deep learning tasks (e.g., an image classification) where the whole corpus is from the same context.

Hyper-parameters tuning. We use the following hyper-parameter when fine-tuning our DeepLineDP models: Bi-GRU hidden cells of token encoder = $\{32, 64, 128\}$, Bi-GRU hidden cells of line encoder = $\{32, 64, 128\}$, MLP hidden cells of token attention = $\{64, 128\}$, MLP hidden cells of line attention = $\{64, 128\}$, learning rate = $\{0.01, 0.001, 0.0001\}$. During the hyper parameter calibration phase, we start with Bi-GRU hidden cells of token encoder = 64, Bi-GRU hidden cells of line encoder = 64, MLP hidden cells of token attention = 64, MLP hidden cells of line attention = 64 and learning rate = 0.001. We then randomly calibrate each hyper parameter, one at a time, to observe the change of model's performance on the validation set. Then, we selected the best model based on the validation set, not the testing set.

Evaluation Measures. We use traditional measures (i.e., AUC, Balanced Accuracy, MCC) for evaluating file-level defect prediction and effort-aware measures (i.e., Recall@Top20%LOC, Effort@Top20%Recall, Initial False Alarms) for evaluating line-level defect prediction.

AUC is an area under the ROC Curve (i.e., the true positive rate and the false positive rate). AUC values range from 0 to 1, with a value of 1 indicates perfect discrimination, while a value of 0.5 indicates random guessing.

Balanced Accuracy measures the average of True Positive Rate (i.e., how many predicted defective files are correct) and True Negative Rate (i.e., how many predicted clean files are correct). Balanced accuracy is computed as: $(\frac{\text{TP}}{\text{TP+FN}} + \frac{\text{TN}}{\text{TN+FP}})/2$, where TP, TN, FP and FN refer to True Positive, True Negative, False Positive, and False Negative, respectively. A high balanced accuracy indicates that an approach can accurately predict defective files and clean files.

Matthews Correlation Coefficients (MCC) measures a correlation coefficients between actual and predicted outcomes using the following calculation:

$$\frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

.

An MCC value ranges from -1 to +1, where an MCC value of 1 indicates a perfect prediction, and -1 indicates total disagreement between the prediction.

Recall@Top20%LOC measures how many defective lines that can be accurately found when inspecting the top 20% LOC of the whole release. A high value of Recall@Top20%LOC indicates that our approach can rank many actual defective lines at the top and many actual defective lines can be found given a fixed amount of effort. On the other hand, a low value of Recall@Top20%LOC indicates that many clean lines are in the top-20% LOC and developers need to spend more effort to identify defective lines.

Effort@Top20%Recall measures how much effort (i.e., LOC) required to find the 20% actual defective lines of the whole release. A low value of Effort@Top20%Recall indicates that developers spend a small amount of effort to find the top-20% actual defective lines. On the other hand, a high value of Effort@Top20%Recall indicates that developers spend a large amount of effort to find the top-20% actual defective lines.

Initial False Alarm measures the number of clean lines that developers need to inspect until the first defective line is found for each file [80]. A low IFA value indicates that few clean lines are ranked at the top, while a high IFA value indicates that developers will spend unnecessary effort on clean lines. The intu-

ition behinds this measure is that developers may stop inspecting if they could not get promising results (i.e., find defective lines) within the first few inspected lines [115].

Note that we do not measure AUC, MCC and Balanced Accuracy for line-level defect predictions since we formulate line-level prediction as a ranking task. In contrast, AUC, MCC and Balanced Accuracy measures are designed for classification tasks. Thus, these measures are not applicable to our line-level defect prediction.

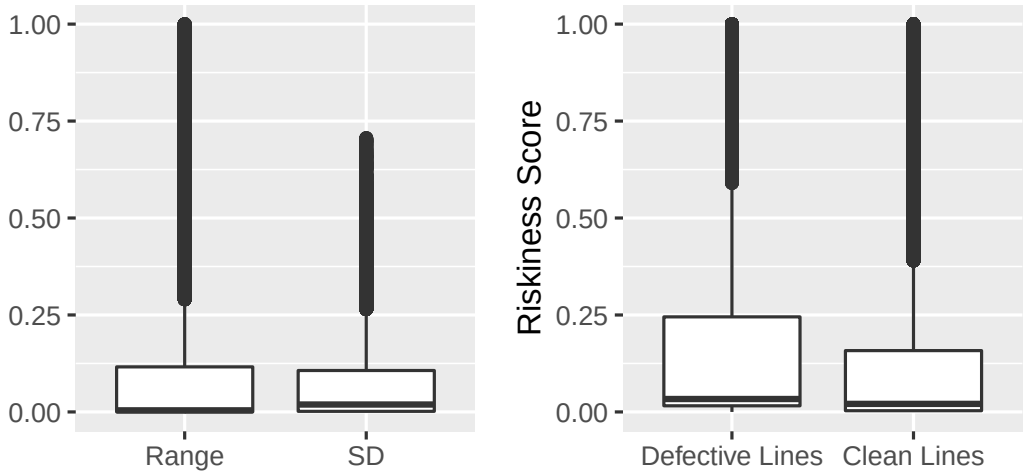
Below, we present the approach and results of our four research questions (RQs).

(RQ1) Can our DeepLineDP be used to differentiate the riskiness of code tokens in defective and clean lines?

Motivation. Not all code tokens and all lines in a defective file are defective. A code token that appears in different lines may have different riskiness. Thus, we investigate if the risk score of code tokens varies depending on their locations.

Approach. We aim to investigate if the risk score of the same code token at different lines varies depending on their locations. To answer this RQ, we use the attention scores calculated by the attention mechanism as a proxy to indicate the risk of code tokens that contribute to the prediction of that defective file. We hypothesize the same code token at different lines should have different riskiness. Thus, we first analyze the variation of the risk scores of each unique code token that appears in different lines in a file using the range (i.e., max-min) and the standard deviation (S.D.). Finally, we analyze the distribution of the risk scores of code tokens that appear in both defective lines and clean lines.

Results. **The risk score of the same code token that appears in different lines varies greatly depending on their locations.** Figure 21a shows that the range of the risk scores varies from 0 to 1, where its standard deviation varies from 0 to 0.73. A range of 1 indicates that the risk score of a code token may be extremely risky in one line, while not being risky at all in another line. We find that there are 26% of code tokens that their risk score varies more than 5% when appearing in different lines, meaning that the risk score of such code tokens varies depending on their locations. This finding suggests that our DeepLineDP approach can be used to differentiate the riskiness of code tokens in defective and clean lines.



(a) The distribution of the range (i.e., Max-Min) of the risk scores of code tokens in a file.

(b) The distribution of the risk scores between actual defective lines and actual clean lines.

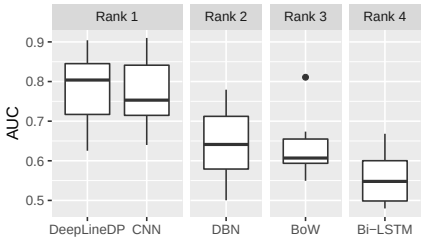
Figure 21: (RQ1) The variation of the risk scores of code tokens in defective lines and clean lines.

In addition, the risk score of code tokens in defective lines is significantly higher than the risk score of code tokens in clean lines. Figure 21b presents the distribution of the risk score of code tokens that appear between actual defective lines and actual clean lines. The Mann-Whitney U test confirms that the risk score of code tokens in defective lines is significantly higher than the risk score of code tokens in clean lines (p-value < 0.001) with a Cliff's δ effect size of small (δ = 0.18-0.20).

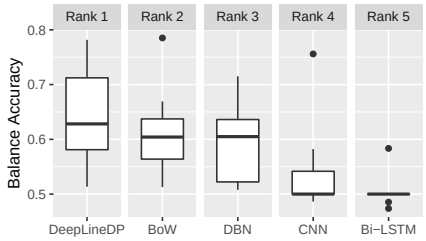
(RQ2) What is the accuracy of our DeepLineDP for predicting defective files?

Motivation. The state-of-the-art file-level defect prediction approaches [97–99, 102] do not consider the surrounding code lines when predicting defective files. Thus, we investigate if our DeepLineDP outperforms the state-of-the-art file-level defect prediction.

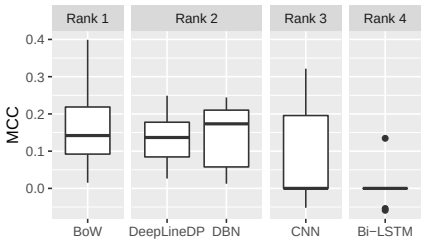
Approach. To answer this RQ, we evaluate our DeepLineDP using cross-release evaluation setting for each project (i.e., the first release R_1 is used for training, R_2 is used for validation, while the subsequent releases $R_3, R_4, ..., R_n$ are used for testing). Therefore, we have a total of 14 train-test evaluation combinations (i.e., 3 for ActiveMQ + 2 for Camel + 2 for JRuby + 2 for Lucene + 1$\times$5 for Derby, Groovy, Hbase, Hive and Wicket). Then, we compare our DeepLineDP with the



(a) AUC-ROC (↗).



(b) Balanced Accuracy (↗).



(c) Mathew Correlation Coefficients (↗).

Figure 22: (RQ2) The ScottKnott ESD ranking and the distributions of the AUC, Balanced Accuracy, and Mathew Correlation Coefficients (MCC) of our DeepLineDP and the state-of-the-art file-level defect prediction approaches. The higher (↗) the values are, the better the approach is.

four state-of-the-art file-level defect prediction approaches:

1. BoW uses the frequency of the code tokens (aka. Bag-of-Word) as textual features to predict defective files [116]. Similar to prior work, we use a logistic regression classification technique to train the BoW features. To handle the class imbalance problem, we applied the SMOTE technique [64] to rebalance the training datasets.
2. DBN is a Deep Belief Networks architecture that automatically learns semantic features for predicting defective files [98, 102].
3. CNN is a Convolutional Neural Network (CNN) architecture that automatically learns semantic and structural features to predict defective files [117, 118].
4. BiLSTM is a bidirectional Long Short-Term Memory architecture that automatically learns semantic and syntactic features to predict defective files [97, 119–121].

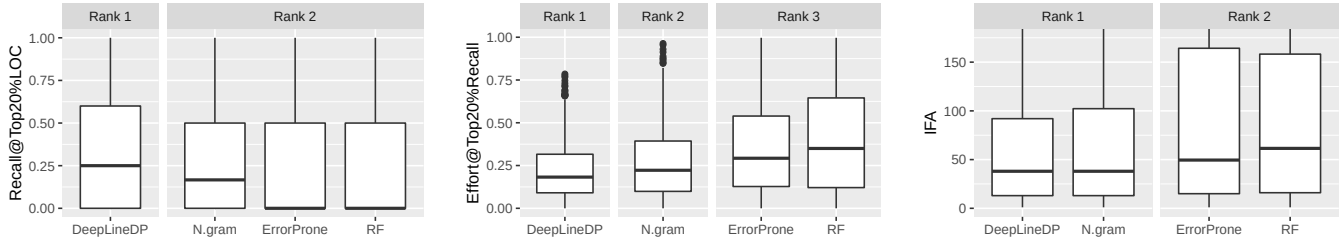
Then, we evaluate these approaches using the 3 traditional evaluation measures (i.e., AUC-ROC, Balanced Accuracy, and Matthews Correlation Coefficients (MCC)). We compute the percentage improvement (%) as follows: $\frac{(P_{\text{ours}} - P_{\text{baseline}})}{P_{\text{baseline}}} \times 100\%$. Finally, we apply a ScottKnott ESD test to cluster the distributions into statistically distinct ranks with non-negligible effect size difference [56, 72, 73]. The ScottKnott ESD test produces the ranking of the techniques while ensuring that (1) the magnitude of the difference for all of the distributions in each rank is negligible; and (2) the magnitude of the difference of distributions

between ranks is non-negligible. Figure 22 presents the ScottKnott ESD ranking and the distributions of the AUC, Balanced Accuracy, and Mathew Correlation Coefficients (MCC) of our DeepLineDP approach and the state-of-the-art file-level defect prediction approaches.

Results. **Our DeepLineDP is 17%-37% and 3%-26% more accurate than the state-of-the-art in terms of the median AUC and the median Balanced Accuracy**. According to the result, the median of AUC is at 0.81 for our DeepLineDP, 0.75 for CNN, 0.64 for DBN, 0.61 for BoW, and 0.55 for BiLSTM models. We also observe a median Balanced Accuracy at 0.63 for our DeepLineDP, 0.61 for DBN, 0.61 for BoW, 0.50 for CNN, and 0.50 for BiLSTM models. These findings indicate that our DeepLineDP approach outperforms the state-of-the-art file-level defect prediction approaches. The ScottKnott ESD test also confirms that our DeepLineDP approach always appears at the top rank in terms of AUC and Balanced Accuracy, indicating that the performance difference is statistically significant with non-negligible effect size. On the other hand, we observed that our DeepLineDP achieves an MCC value slightly lower than the BoW approach, since the MCC measure is designed to capture the False Positive Rates and False Negative Rates, while the Balance Accuracy captures only the True Positive Rate and True Negative Rate. When we perform a deeper investigation in the prediction results, DeepLineDP tends to produce more false positive than the BoW (i.e., incorrectly predicts clean files as defective). Nevertheless, in a deployment scenario, developers often consider the ranking of the defective files (i.e., based on a ranking of probabilities), rather than the binary classification of the defective and clean files (i.e., based on a threshold of 0.5). Thus, when defective files are ranked based on the probabilities, AUC that captures the ability to distinguish defective and clean files still confirm that our DeepLineDP still outperforms other techniques.

(RQ3) What is the cost-effectiveness of our DeepLineDP for locating defective lines?

Motivation. Line-level defect prediction is needed to help developers identify the fine-grained locations of defective code, instead of wasting their time inspecting clean lines. However, there exists only few approaches for prioritizing defective lines [75, 112, 122, 123]. Thus, we investigate if our DeepLineDP that is able to locate defective lines is more cost-effective than the state-of-the-art line-level defect prediction.



(a) Recall@Top20%LOC (↗). (b) Effort@Top20%Recall (↘). (c) Initial False Alarms (↘).

Figure 23: (RQ3) The ScottKnott ESD ranking and the distributions of the Recall@Top20%LOC, Effort@Top20%Recall, and Initial False Alarms of our DeepLineDP and the state-of-the-art line-level defect prediction approaches. The higher (↗) or the lower (↘) the values are, the better the approach is.

Approach. To answer this RQ, we investigate if our DeepLineDP, being able to locate defective lines, is more cost-effective than the state-of-the-art line-level defect prediction. To evaluate the cost-effectiveness of line-level defect prediction, we only focus on the correctly predicted defective files from our models. For each defective file, we first compute the attention scores using the attention mechanism to indicate the risk score of each token. Then, we compute the summation of the attention scores for each line. After that, we generate the ranking of risky lines for each file based on the risk scores. Finally, we compare the ranking of risky lines with the three state-of-the-art line-level defect prediction approaches:

1. N-gram is used to infer the unnaturalness of each code token [75, 112, 122] based on the entropy score of each code token. We selected N-gram as our baseline, since prior studies showed that N-gram is effective in identifying defective lines [112, 122] (i.e., high entropy = unnaturalness of code tokens). We use an implementation of Hellendoorn and Devanbu [75] to build cache-based language models, i.e., an enhanced n-gram model that is suitable for source code. The cache-based language models are built by learning from clean code files (i.e., the files that do not contain defective lines). Similar to prior work [75], once the entropy scores for all code tokens are calculated, the average of the entropy scores for each line is computed. Then, we rank the source code lines based on the average entropy values of tokens in each line.
2. ErrorProne [124] is a Google's static analysis tool that builds on top of a primary Java compiler (javac) to check errors in source code based on a set of error-prone rules. We use ErrorProne instead of other static analysis tools (e.g., FindBugs) since ErrorProne is able to detect more critical software de-

Table 7: (RQ4) The mean performance of our DeepLineDP approach for within-project (i.e., using the models trained using its own project) and cross-project evaluation (i.e., using the models trained using other projects) and its percentage point difference (δ =Delta Difference). The higher (↗) or the lower (↘) the values are, the better the approach is.

	AUC (↗)			Balanced Accuracy (↗)			MCC (↗)			Recall@20%LOC (↗)			Effort@20%Recall (↘)			Initial False Alarms (↘)		
Target	**Within**	**Cross**	δ	**Within**	**Cross**	δ	**Within**	**Cross**	δ	**Within**	**Cross**	δ	**Within**	**Cross**	δ	**Within**	**Cross**	δ
activemq	0.87	0.74	-0.13	0.61	0.66	0.05	0.12	0.20	0.08	0.39	0.31	-0.08	0.20	0.24	0.04	49	69	20
camel	0.76	0.74	-0.02	0.70	0.60	-0.10	0.17	0.13	-0.04	0.30	0.46	0.16	0.27	0.20	-0.07	51	59	8
derby	0.85	0.74	-0.11	0.75	0.56	-0.19	0.25	0.07	-0.18	0.47	0.44	-0.03	0.16	0.16	0.00	156	150	-6
groovy	0.84	0.79	-0.05	0.51	0.68	0.17	0.03	0.16	0.13	0.41	0.37	-0.04	0.16	0.19	0.03	69	97	28
hbase	0.65	0.63	-0.02	0.61	0.56	-0.05	0.14	0.08	-0.06	0.27	0.34	0.07	0.24	0.22	-0.02	125	136	11
hive	0.74	0.69	-0.05	0.67	0.60	-0.07	0.23	0.13	-0.10	0.15	0.36	0.21	0.28	0.20	-0.08	137	92	-45
jruby	0.82	0.68	-0.14	0.75	0.57	-0.18	0.17	0.05	-0.12	0.30	0.36	0.06	0.28	0.24	-0.04	211	205	-6
lucene	0.65	0.63	-0.02	0.57	0.54	-0.03	0.07	0.04	-0.03	0.53	0.39	-0.14	0.15	0.19	0.04	63	55	-8
wicket	0.75	0.78	0.03	0.56	0.58	0.02	0.06	0.08	0.02	0.88	0.69	-0.19	0.10	0.12	0.02	29	42	13

fects (e.g., compilation defects) than FindBugs [124]. In this experiment, we consider lines as defective if errors are generated by ErrorProne. This mimics a top-down reading approach, i.e., developers sequentially read source code from the top to the bottom of the files.

3. Random Forest (RF) is used to predict which lines in a defective file are clean or defective. The model used DeepLineDP's initial token representation at line level as features to train the model.

We again compute the percentage improvement (%) and apply the ScottKnott ESD test to cluster the distributions into statistically distinct ranks with non-negligible difference. Figure 23 presents the ScottKnott ESD ranking and the distributions of the Recall@Top20%LOC, Effort@Top20%Recall, and Initial False Alarms of our DeepLineDP and the state-of-the-art line-level defect prediction approaches.

Results. **Our DeepLineDP is 47% - 250% more cost-effective than the state-of-the-art line-level defect prediction approaches in terms of Recall@Top20% LOC.** We find that our DeepLineDP approach achieves a median Recall@Top20% LOC of 0.25, while N-gram achieves a median Recall@Top20%LOC of 0.17, and ErrorProne and RF achieves a median Recall@Top20%LOC of 0 . This finding indicates that, given a fixed amount of 20%LOC inspection effort, our DeepLineDP approach can correctly locate 25% of the actual defective lines, while N-gram can correctly locate only 17%, and ErrorProne and RF can correctly locate 0% of the actual defective lines.

In addition, our DeepLineDP achieves a median Effort@Top20%Recall of 0.18, which is 38% and 49% less than ErrorProne and N-gram, respectively. We find that our DeepLineDP approach achieves a median Effort@Top20%Recall of

0.18, while N-gram, ErrorProne and RF achieve a median Effort@Top20%Recall of 0.22, 0.29 and 0.35, respectively. This finding indicates that, in order to find the 20% actual defective lines, our DeepLineDP approach approximately requires developers to inspect 18% LOC of the whole release, while the N-gram approximately requires 22%, ErrorProne approximately requires 29% and the RF models approximately requires 35%. However, as mentioned in the threat to construct validity, the reported effort may not reflect the actual effort that a developer requires to inspect defective lines.

When inspecting the ranking of risky lines in each defective file, our DeepLineDP and N-gram achieve a median Initial False Alarm of 38, while the ErrorProne and the RF models achieve a median Initial False Alarm of 50 and 62, respectively. According to this finding, the ranking of risky lines generated by our DeepLineDP is better than ErrorProne and N-gram. Thus, developers will spend less inspection effort to find the first actual defective line.

The ScottKnott ESD test also confirms that our DeepLineDP approach always appears at the top rank in terms of Recall@Top20%LOC, Effort@Top20%Recall, and Initial False Alarms with non-negligible effect size. This finding suggests that the hierarchical structure and code context (i.e. surrounding tokens and surrounding lines) can be used to improve the cost-effectiveness of SQA resource management, while saving developers' inspection effort to locate actual defective lines.

(RQ4) What is the accuracy of our DeepLineDP for line-level cross-project defect predictions?

Motivation. Intuitively, at the beginning of software development, the amount of historical data is limited. Thus, applying defect prediction to such early-development software is often inaccurate. Thus, prior studies suggested to train a model from a source project and test on the target project (i.e., cross-project defect prediction). Yet, Zimmermann *et al.* [125] found that such simple cross-project prediction scenario often produce inaccurate predictions. Thus, prior studies proposed various advanced approaches to improve the performance of cross-project defect prediction at different granularity levels. For example, Menzies *et al.* [126], Zhang *et al.* [127] and Xia *et al.* [128] proposed class-level cross-project defect prediction approaches. Zimmermann *et al.* [125] and Wang *et al.* [102] proposed file-level cross-project defect prediction approaches. Turhan *et al.* [129] proposed module-level cross-project defect prediction approaches. Li *et al.* [118] and Dam *et al.* [97]

proposed method-level cross-project defect prediction approaches. However, none of the prior cross-project defect prediction approaches can predict defects at the line level. Thus, we investigate how accurate our DeepLineDP model is for line-level cross-project defect predictions.

Approach. To answer this RQ, we apply the DeepLineDP models trained on a given source project to the releases of the target projects other than the source project. Next, we evaluate the performance of file-level defect prediction using 3 traditional measures and the performance of line-level defect prediction using 3 effort-aware measures. Then, we compare the performance of cross-project predictions with within-project predictions. In total, we have a combination of 112 train-test evaluation settings. Table 7 presents the median performance of our DeepLineDP approach for within-project (i.e., the models trained using its own project) and cross-project evaluation (i.e., the models trained using other projects) and its percentage point difference (δ =Delta Difference).

Results. **Our DeepLineDP models are transferable to other software projects, since our DeepLineDP models achieve a reasonable AUC of 0.63-0.79.** Table 7 shows that, when the models are trained using other projects, our DeepLineDP models achieve an AUC of 0.63-0.79, a Balanced Accuracy of 0.54-0.68, an MCC of 0.07-0.16. However, the AUC performance of cross-project predictions is still comparable with within-project defect predictions, with a percentage point difference between -0.14 and +0.03. Even though we find that the AUC of the models trained using other projects can be less accurate than the models trained using its own project, we still achieve a reasonable AUC for file-level defect prediction.

Our DeepLineDP models achieve a reasonable Recall@Top20%LOC of 0.31-0.46, an Effort@Top20%Recall of 0.12-0.24, and an Initial False Alarm of 42-205. Table 7 shows that, when the models trained using other projects, our DeepLineDP can still correctly locate 31%-46% of the actual defective lines when inspecting the top 20% LOC. We also find that our DeepLineDP approximately requires 12%-24% LOC effort to find 20% of the actual defective lines. However, our DeepLineDP approach can be more or less cost-effective than the models trained using its own project, since the Recall@Top20%LOC varies between -0.19 and +0.21 when compared to within-project defect predictions. Nevertheless, we still achieve a reasonable Recall@Top20%LOC, Effort@Top20%Recall and an Initial False Alarm for line-level defect prediction. These findings suggest that developers in different software projects may use our DeepLineDP models trained from other projects to identify the fine-grained locations of defective code

(i.e., defective files and defective lines).

4.5 Discussion

In this section, we provide additional discussions and disclose the threats to the validity.

4.5.1 Implications to Practitioners

Our motivating survey found that 50% of the respondents spend at least 10 minutes to more than one hour to review a single file, while 64% of the respondents perceived that current code inspection activity is very challenging to extremely challenging. Importantly, 44% of the respondents perceived that a line-level defect prediction tool would potentially be helpful in identifying defective lines, and 64% of the respondents would consider using a line-level defect prediction tool if it is publicly available for free. Our results show that our DeepLineDP can accurately predict defective files achieving an AUC of 0.63-0.91, while also accurately predicting defective lines achieving a Recall@Top20%LOC of 0.25, which is 50%-250% more cost-effective than the state-of-the-art approaches. Since the data collection steps and the model training can be automated and the model implementation cost is minimal (e.g., 5 minutes for model training time on a consumer-grade GPU), we believe that the cost of adoption of our approach in software industry would be very minimal.

4.5.2 DeepLineDP vs Code Coverage

Generally, code coverage often provides sufficient information for developers to prioritize testing resources on the lines that are not covered by the test cases. Particularly, code coverage is used to indicate the proportion of lines of code that are tested and not tested when executing a set of test suites (i.e, white-box testing), indicating that lines of code that are not executed by test cases may be more defect-prone than the others.

Nevertheless, a software project that achieves 100% code coverage through white-box testing only indicates that the 100% of these lines of code have been executed by the test cases, but it does not always mean that these lines of code will not be defective in the future (i.e., the presence of bugs do not indicate the absence of bugs, since the absent bugs may appear in the future). Similarly,

DeepLineDP alone may help developers detect areas that are likely to be defective in the future, but not the presence of defects. Thus, DeepLineDP and code coverage should be used in complementary in order to achieve the highest quality of software systems.

4.5.3 Threats to Validity

Threats to construct validity relate to the suitability of our evaluation measure. We use only three traditional measures that are less susceptible to class imbalance problems [63] and use three effort-aware measures that are prefered by practitioners [130]. However, other measures can be used for evaluation as well.

Similar to prior work [63, 74, 79, 131], we use LOC as a proxy to measure developers' effort. However, LOC may not reflect actual effort that a developer requires to inspect defective lines. Two files with the same amount of lines of code may have different complexity levels, which requires different amounts of effort to inspect. Thus, other factors like code complexity could be incorporated into the consideration in future work. In addition, these effort measures are based on the assumption that developers inspect defective lines independently. However, as confirmed by our survey, developers often have various code inspection practices (e.g., a top-down order or a random order). Thus, future work may conduct an observational study with industrial practitioners to better understand how do practitioners perform code inspection in real-world practices.

Various cross-project defect prediction approaches have been proposed to improve the cross-project defection performance (e.g., data clustering, data normalization). However, such approaches are not designed for predictions at the line level. Thus, we do not perform any data preprocessing for our line-level cross-project defect prediction, since such file-level cross-project defect prediction approaches are specifically designed for software metrics and machine learning approaches, not for embedding vectors and deep learning.

As suggested by Rahman and Rigby [83], we removed special characters. However, it is possible that such special characters may contribute to software defects (e.g., missing parentheses). Nevertheless, the design of our DeepLineDP approach is focused on the semantic properties of source code (i.e., input sequences of source code), not the syntactic properties. Thus, other information like syntactic could be considered in future work to improve the performance.

Threats to external validity concern the generalizability of our work. Our results are limited to the studied 32 software releases from 9 software projects. Thus, our results may not generalize to other software projects.

In addition, DeepLineDP is specifically designed for release-based defect prediction models. Thus, DeepLineDP is not applicable to other types of defect prediction models (e.g., Just-In-Time defect prediction). This is due to the different structure of information. For release-based defect prediction, DeepLineDP is designed to capture the structure of source code (i.e., tokens forming lines, lines forming files). On the other hand, for commit-based defect prediction, the structure of code changes is that tokens forming lines, lines forming changed hunks, hunks forming changed files, changed files forming one commit, which requires a different neural network architecture.

Threats to internal validity concern the impact of the hyperparameter settings on the performance of our DeepLineDP models. The performance of our models would be affected by the different weights, which are tuned by hand in our experiments. However, optimizing the parameter settings for deep learning approaches is computationally expensive.

4.6 Related Work

Below, we discussed the related work regarding defect prediction approaches in different aspects.

4.6.1 Granularity Levels of Defect Prediction Models

In the past decades, prior studies proposed defect prediction approaches at various granularity, e.g., packages [74], components [40], modules [100], files [57, 58, 74, 79, 132], methods [101], and commits [25]). However, developers still waste a large amount of SQA effort on locating actual defective lines. Recently, a survey by Wan *et al.* [130] found that practitioners prefer fine-grained defect prediction. Thus, a line-level defect prediction is needed to help developers prioritize their SQA effort on the fine-grained high-risk areas of source code in a cost-effective.

4.6.2 Software Features for Defect Prediction

Traditionally, the accuracy of defect prediction models heavily relies on software features. There are numerous static source code attributes [133] that share a

strong relationship with software quality,. For example, file size [134], McCabe's Cyclomatic code complexity [135], code smells [136], and test smells [137]. In addition, software development practices may share a strong relationship with software quality. For example, organization structure [138], code ownership [139–141], the entropy of change [142, 143], and developers' characteristics [144]. However, these software features are specifically designed for file-level defect prediction. Thus, the approaches that rely on these software features cannot predict defect at line level.

4.6.3 Deep Learning Approaches for Defect Prediction

Wang *et al.* [98] used a Deep Belief Network (DBN) architecture to represent a source code file using semantic features. Li *et al.* [117, 118] used a Convolutional Neural Network (CNN) architecture to learn the semantic and structural features of source code. Dam *et al.* [97] used a Long Short-Term Memory (LSTM) architecture to learn the semantic and syntactic features of source code. Zou *et al.* [120] and Li *et al.* [121] used a bidirectional Long Short-Term Memory architecture to learn semantic and syntactic features to predict defective files. However, the granularity level of predictions of prior deep learning-based defect prediction models is still coarse-grained (i.e., file level). Different from prior work, we present a fine-grained deep learning-based defect prediction approach that can predict defects at line level.

4.6.4 Line-Level Defect Prediction

Prior studies attempt to predict defective lines using various approaches [28, 45, 111, 112, 122, 145]. The simplest approach is to apply static analysis tools to identify defective lines based on a set of predefined rules. However, static analysis tools generate too many false positives warnings [146], and they may not be associated with post-release defects. Recently, Majd *et al.* [145] proposed a suite of 32 C/C++-based statement-level features (e.g., the number of binary and unary operators used in a statement) for statement-level defect prediction using a Long Short-Term Memory (LSTM) neural network architecture. However, these statement-level features only capture the static attributes of statements, which still require handcrafted features extracted by data scientists. Ray *et al.* [112] and Wang *et al.* [122] proposed to use an n-gram language model to predict the unnatural code tokens, which later can be used to identify defective lines. While the n-gram language model can capture the surrounding code tokens, its length of surrounding code tokens is limited to n tokens. Recently, studies leveraged a model-

agnostic technique (i.e., LIME) from the Explainable AI domain to develop line-level defect prediction for just-in-time defect prediction [28] and release-based defect prediction [45]. Unlike existing techniques, our DeepLineDP approach is the first deep learning approach for line-level defect prediction that automatically extracting features (i.e., semantic properties).

4.6.5 Explainable AI for Software Engineering

Explainable AI has been actively investigated in the domain of defect prediction [34, 35]. Recently, Jiarpakdee *et al.* [33] found that explaining the predictions are as equally important and useful as improving the accuracy of defect prediction. However, their literature review found that 91% (81/96) of the defect prediction studies only focus on improving the predictive accuracy, without considering explaining the predictions, while only 4% of these 96 studies focus on explaining the predictions.

Although Explainable AI for SE is still very under-researched, recent works have shown some successful case studies to make defect prediction models more practical [28, 45], explainable [32, 39], and actionable [50, 147]. For example, Pornprasit and Tantithamthavorn [28] leveraged LIME for line-level just-in-time defect prediction. Wattanakriengkrai *et al.* [45] leveraged LIME for line-level release-based defect prediction, helping developers better identify the location of software defects. Jiarpakdee *et al.* [32] and Khanan *et al.* [39] employed model-agnostic techniques (e.g., LIME) for explaining defect prediction models, helping developers better understand why a file is predicted as defective. Rajapaksha *et al.* [50] and Pornprasit *et al.* [147] proposed local rule-based model-agnostic techniques to generate actionable guidance to help managers chart the most effective quality improvement plans. However, these studies only focus on explaining the traditional machine learning approaches. Unlike prior studies, our DeepLineDP is a deep learning approach that is designed to be interpretable by using the attention mechanism. We also make use of the attention mechanism to identify the most risky tokens and risky lines.

4.7 Summary

In this chapter, we present DeepLineDP, a deep learning approach to automatically learn the semantic properties of the surrounding tokens and lines in order to identify defective files and defective lines. Through a case study of 32 releases of 9 software projects, we find that the risk score of code tokens varies greatly

depending on their location. Our DeepLineDP is 14%-24% more accurate than other file-level defect prediction approaches; is 50%-250% more cost-effective than other line-level defect prediction approaches; and achieves a reasonable performance when transferred to other software projects.

5 D-ACT: Towards Diff-Aware Code Transformation for Code Review Under a Time-Wise Evaluation

Chapter Overview

Practitioners usually face the challenge of receiving timely feedback when performing code review. Thus, prior works have proposed code transformation approaches to help practitioners automate code review activity. However, such approaches did not utilize contextual information in learning code transformation. Therefore, such approaches may make changes to code that should remain unchanged or recommend code that has different coding conventions.

In this chapter, to address the overarching research question of the thesis, we propose D-ACT that incorporates a pre-trained code model and token-level code difference information, obtained from the previous version of a patch and the first version of the patch, to better recommend potential code improvement of the first version of the patch. Through the study with three large-scale open-source software projects (i.e., Android, Google and Ovirt) we find that token-level code difference information in our D-ACT approach can substantially improve the performance of NMT-based code transformation approaches for code review.

The work in this chapter appears in Chanathip Pornprasit, Chakkrit Tantithamthavorn, Patanamon Thongtanunam and Chunyang Chen.“D-ACT: Towards Diff-Aware Code Transformation for Code Review Under a Time-Wise Evaluation.” In Proceedings of SANER. 2023, pp. 296–307.

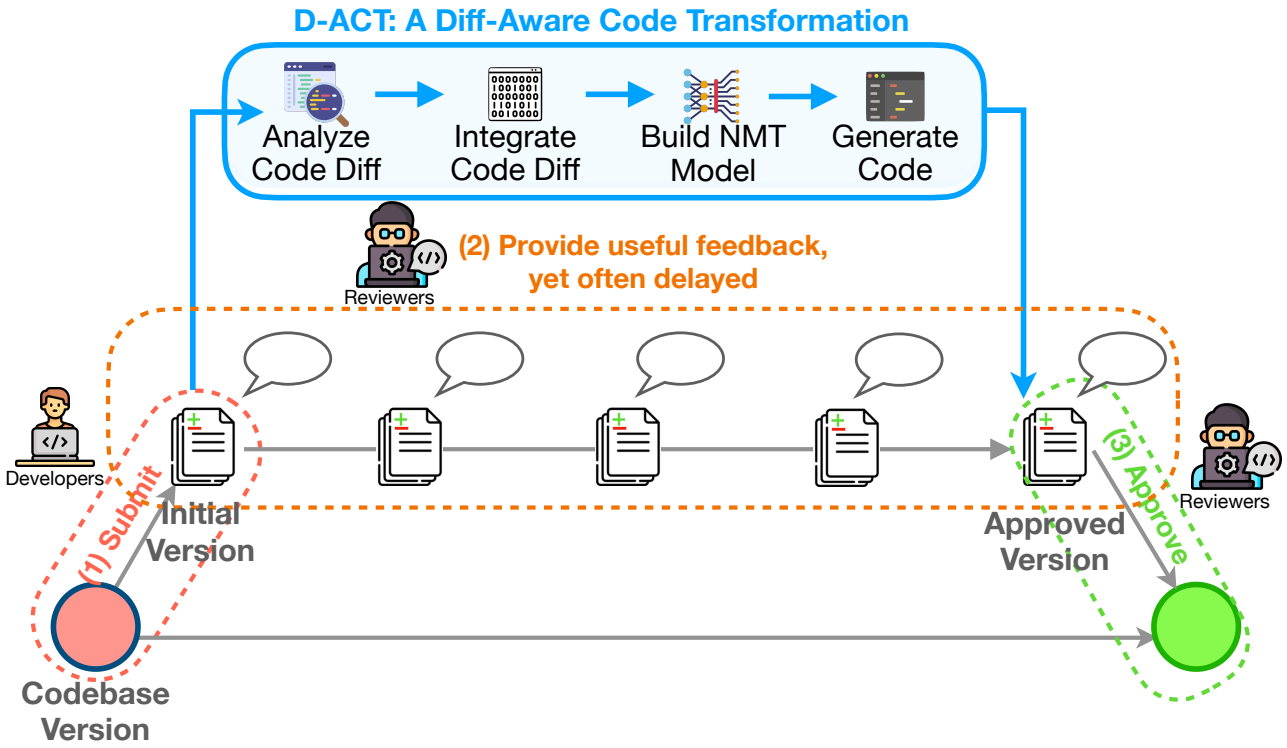


Figure 24: A usage scenario of our Diff-Aware Code Transformation (D-ACT) approach.

5.1 Introduction

Code review is the practice where reviewers (i.e., developers other than the author) review a *patch (i.e., a set of code changes)* to ensure that it meets quality standards before being approved to be integrated into a codebase. Recent studies showed that developers perform code review to identify security issues [148]; and detect code smells [149], issues of refactored code [3], and software defects [19, 150]. Although code review brings a lot of benefits to software development, developers still face challenges while revising their submitted patches. In particular, one of the top challenges of developers is receiving timely feedback [150]. Rigby *et al.* [5] also reported that developers may have to wait for 15 to 20 hours to receive feedback for their initial version of the submitted patches.

Neural Machine Translation (NMT)-based code transformation approaches have been proposed to facilitate developers revising their submitted patches. For example, Thongtanunam *et al.* [14] proposed AutoTransform, which leverages a Transformer model [105] and Byte-Pair Encoding (BPE) subword tokenization [151] for code transformation. Similarly, Tufano *et al.* [24] proposed a T5-based pre-trained code model for automated code review activities (called, TufanoT5 onwards).

Intuitively, the changed code tokens in the method should be paid more attention to than the others as they are more prone to be defective [152]. In addition, prior work found that changed code in the past is more likely to be changed again in the future (e.g., fixing software defects) [153]. Unfortunately, the existing code transformation approaches [13–15, 24] only learn the sequence of code tokens **without knowing which code tokens are previously changed from the codebase**.

In this chapter, we present a Diff-Aware Code Transformation (D-ACT) approach for code review that leverages the token-level code difference information and CodeT5 [26] (the state-of-the-art pre-trained code model). Different from the existing approaches [14, 24], we design our D-ACT approach to transform *the initial version* (i.e., the first version of a submitted patch) to *the approved version* (i.e., the version after being reviewed and approved by reviewers) while considering the *token-level code difference information* which is extracted from the code difference between the codebase version and the initial version. Our intuition is that changed code tokens are more likely to introduce defects than the others [152]; thus, explicitly providing such information may help the NMT models better learn the code transformation. While temporal information is well-regarded in the literature [28, 89, 93, 154], such temporal information remains largely ignored by the existing code transformation approaches for code review [14, 24]. Thus, we are the first to evaluate the NMT-based code transformation approaches for code reviews under the time-wise scenario. Specifically, we consider the chronological order of patches in our evaluation, meaning that no future patches are used to train the models. Finally, we conduct experiments based on our datasets collected from three large-scale software projects, i.e., Google, Ovirt, and Android. Through an experimental evaluation of 57,615 changed methods that span across the three software projects, we address the following research questions:

(RQ1) How do our approach and state-of-the-art code transformation approaches for code review perform under the time-wise evaluation scenario?
Result. Our D-ACT can correctly transform 107-245 changed methods (i.e., beam size = 10), which is at least 62% higher than TufanoT5 [24] and 274% higher than AutoTransform [14]. Besides, when the state-of-the-art code transformation approaches (i.e., TufanoT5 and AutoTransform) are evaluated under the time-wise scenario, the number of perfect match decreases by at least 57% and 92%, respectively.

(RQ2) What are the contributions of the components (i.e., the token-level

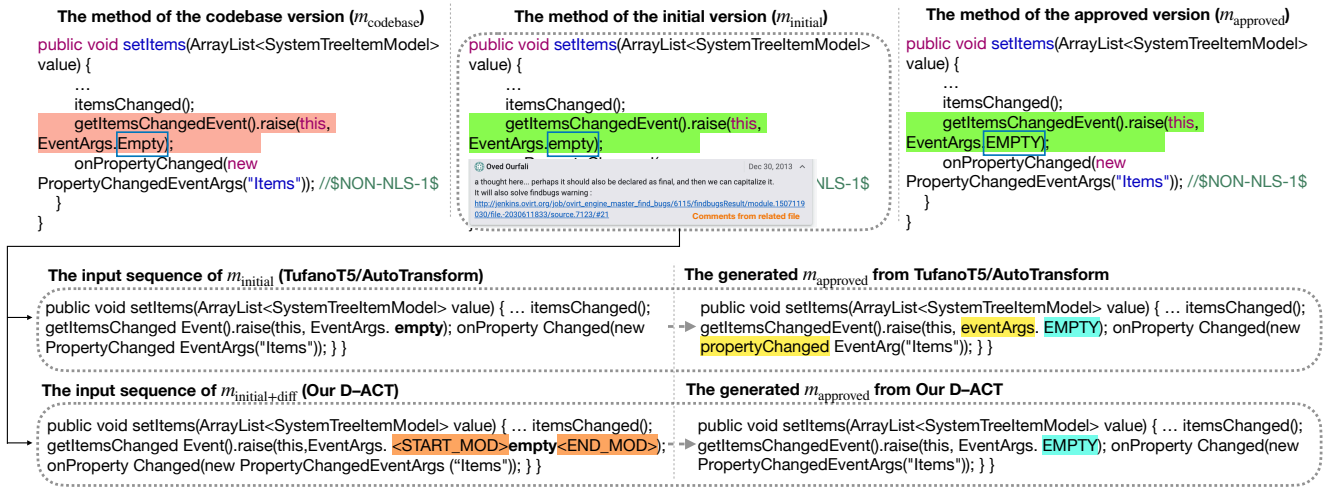


Figure 25: A real-world example of a code review from the Ovirt project. As shown in this example, reviewers tend to provide comments related to the changed code tokens (i.e., `empty`) rather than the others where this token is eventually changed to `EMPTY` in the approved version. However, the existing code transformation approaches for code review may make changes to the code tokens that should remain unchanged (highlighted in yellow), since these approaches do not know which code tokens should be paid more attention to.

code difference information and the pre-trained model) of our approach?
Result. Using the token-level code difference information can increase the number of correct transformations by 17%-82% when compared to without using such information. In addition, using CodeT5 can increase the number of correct transformations by at least 29% when compared to using the other pre-trained code models (i.e., CodeBERT [155], GraphCodeBERT [156], PLBART [157]).

In summary, these results suggest that (1) NMT-based code transformation approaches for code review should be evaluated under the time-wise evaluation; and (2) the token-level code difference information can substantially improve the performance of NMT-based code transformation approaches for code review.

Open Science. To support future studies, our replication package is available in GitHub [158].

Paper Organization. Section 5.2 describes the background and related work in order to situate this paper with respect to the literature. Section 5.3 presents our proposed D-ACT. Section 5.4 describes the experimental design. Section 5.5 presents the experimental results of our study. Section 5.6 discusses our experimental results. Section 5.7 discloses possible threats to the validity. Section 5.8 draws the conclusions.

5.2 Background & Related Work

In this section, we describe the background and related work to situate the paper with respect to the literature, and describe the limitations of state-of-the-art code transformation approaches for code review.

5.2.1 Code Review

Code review is a software quality assurance practice where developers other than patch authors are required to provide feedback to artifacts to ensure that the artifacts meet quality standards. Nowadays, code review practices are commonly performed in asynchronous tool-based settings (e.g., Gerrit, GitHub, Phabricator, Review Board) [2, 5]. Figure 24 presents an overview of a code review process, which comprises three main steps. In Step 1., developers (i.e., patch authors) create the first version of a patch (i.e., the *initial version*) by modifying source code in the codebase (i.e., the *codebase version*) or creating the new one. Then, developers submit the created patch to a code review platform (e.g., Gerrit). Next, in Step 2, reviewers review and provide feedback on the submitted patch. Then, the developers revise the patch and submit it to the code review platform again. Finally, in Step 3, if reviewers agree that the recent version of the submitted patch can be integrated into the codebase, they will approve that recent version of the submitted patch (i.e., the *approved version*) to be integrated into the codebase.

5.2.2 Automated Code Transformation for Code Review

Although different automated approaches were proposed to support code review activities (e.g., review prioritization [159, 160], just-in-time defect prediction [28, 29, 39] and localization [45, 147, 161, 162], reviewer recommendation [163–167], AI-assisted code reviewer [168, 169]), such activities still require manual effort which can be time-consuming for developers [19, 150].

Indeed, prior work reported that reviewers could spend a large amount of time on reviewing code (e.g., developers in open-source software projects have to spend more than six hours per week on average reviewing code [6]). Furthermore, it is also challenging for patch authors to receive timely feedback. For example, developers at Microsoft may wait for 15 to 20 hours to receive feedback for their first version of the submitted patches [5]. Therefore, an automated code transformation approach could potentially save developers' effort by automatically revising submitted patches.

Neural Machine Translation (NMT)-based code transformation approaches have been proposed to facilitate developers by generating an approved version of a submitted patch. Broadly speaking, the NMT-based code transformation approaches will transform the method of the initial version m_{initial} in a submitted patch to the method of the approved version m_{approved}. To do so, such approaches learn the mapping between m_{initial} and m_{approved} by computing the conditional probability $p(m_{\text{approved}} \mid m_{\text{initial}})$.

Recent studies have proposed various NMT-based code transformation approaches for code review. Tufano *et al.* [13] were the first to leverage the RNN architecture and code abstraction to learn meaningful code changes in code review. Later, Thongtanunam *et al.* [14] presented AutoTransform to address the limitation of Tufano *et al.*'s approach [13], by leveraging a Byte-Pair Encoding (BPE) subword tokenization [151] and a Vanilla Transformer architecture [105] to better handle the changed methods where the approved versions contain newly-introduced code tokens.

However, such approaches may incorrectly transform m_{initial} to m_{approved} since the models are trained from limited knowledge of source code (i.e., a limited amount of training data).

Recently, Tufano *et al.* [24] proposed to build a pre-trained language model using the Text-to-Text Transfer Transformer (T5) architecture [170] in order to learn general knowledge of source code (called *TufanoT5*, henceforth). By using a transfer learning approach [171], the model is then fine-tuned on the code review dataset to perform the code transformation task. With the use of a transfer learning approach (i.e., pre-trained and then fine-tuned), the model is able to generate better vector representation, producing more accurate generations of source code for the code transformation task. Although prior studies have shown promising results of the previous NMT-based code transformation approaches for code reviews [14, 24], the performance of NMT-based code transformation approaches heavily relies on the knowledge that is used to train a model and the methodology that is used to evaluate the model.

Lack of Code-Diff Information. As pointed out by Beller *et al.* [152], code changes are prone to be more defective than others, thus the area of code changes should require more attention than the others. In addition, Xie *et al.* [153] also found that changed code in the past is more likely to be changed again in the future (e.g., fixing software defects). However, the existing code transformation

approaches [13–15, 24] only learn the sequence of code tokens without knowing which code tokens are previously changed from the codebase. Therefore, it is possible that the existing code transformation approaches may transform code tokens that should remain unchanged and also may not transform code tokens that should be changed, as pointed out by Thongtanunam *et al.* [14].

To illustrate this scenario, Figure 25 presents a real–world example of a code review from the Ovirt project. Generally, the existing code transformation approaches require a sequence of code tokens in the initial version without knowing that `empty` is the only token that was changed from the codebase version. With the existing code transformation approaches, they may transform the code tokens `EventArgs` and `PropertyChanged` to `eventArgs` and `propertyChanged` (highlighted in yellow) in the generated approved version, while these two code tokens should remain unchanged in the actual approved version. To address this challenge, we set out to investigate the impact of the token-level code difference information on the performance of automated code transformation approaches.

Lack of Temporal Information. Code review is a practice that is conducted in temporal order. Thus, temporal information must be considered in the evaluation setup. Prior studies raised concerns that the evaluation setup may have a negative impact on the performance of automated approaches for software engineering tasks (called *experimental bias*). For example, prior work [28, 93] found that temporal information must be considered when evaluating the just-in-time defect prediction models. Similarly, Jimenez *et al.* [89] also found that unrealistic labelling due to a lack of temporal information has a negative impact on the performance of vulnerability prediction approaches. Liu *et al.* [154, 172] also found that by changing the experimental setup to consider temporal information, the performance of ML-based malware detection approaches is substantially decreased. While temporal information is well-regarded in the literature, such temporal information remains largely ignored by the existing code transformation approaches for code review [13–15, 24]. Thus, it is possible that these approaches may learn some of the future patches to generate an approved version of the methods in old patches, which is not well aligned with the realistic evaluation scenario. To address this challenge, we set out to investigate the impact of the time-wise scenario on the performance of automated code transformation approaches.

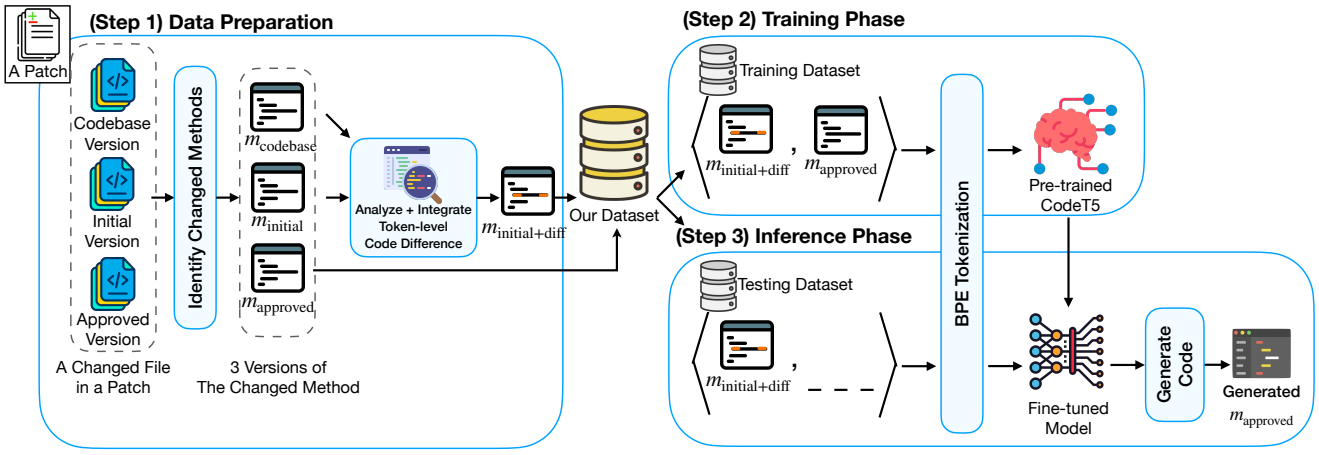


Figure 26: An overview of our approach.

5.3 D-ACT: Diff-Aware Code Transformation for Code Review

In this section, we present our D-ACT, an NMT-based code transformation approach that leverages the token-level code difference information and CodeT5 — the state-of-the-art pre-trained code model [26].

Our underlying intuition of using the token-level code difference information is that changed code tokens are more likely to introduce defects than the others [152]; thus, explicitly providing such information may help NMT models better learn the code transformation. Figure 25 illustrates an example of the input of our approach. Generally speaking, D-ACT uses special tokens (i.e., `<START_MOD>` and `<END_MOD>`) to explicitly specify which code tokens in the initial version were changed from the codebase version. We hypothesize that with this information, the NMT model will better capture the relationship between all code tokens and the given special tokens, reducing the probability that code tokens outside of the special tokens will be changed. We also opt to use the CodeT5 pre-trained model because the model was specifically trained to learn the syntactic and semantic information of code tokens. This is different from the pre-trained T5 model of Tufano *et al.* [24], which learns source code without focusing syntactic information of code tokens.

5.3.1 Overview

Figure 26 presents an overview of our D-ACT approach, which consists of three main steps: (Step 1) data preparation, (Step 2) training phase, and (Step 3) inference phase. In Step 1, we extract three versions of a changed method, i.e., codebase (m_{codebase}), initial (m_{initial}), and approved versions (m_{approved}). Then, we

integrate the token-level code difference information into m_{initial} (i.e., $m_{\text{initial+diff}}$). Once we prepare the dataset, we use pairs of $m_{\text{initial+diff}}$ and m_{approved} to fine-tune the CodeT5 pre-trained model in Step 2. Finally, in Step 3, for each patch in the testing dataset, we use the fine-tuned CodeT5 model to generate m_{approved} based on a given $m_{\text{initial+diff}}$. We describe the details of each step of our D-ACT below.

5.3.2 (Step 1) Data Preparation

The key goal of our data preparation step is to integrate the token-level code difference information into the m_{initial}. To do so, for each changed file in each patch, we first identify changed methods based on its codebase, initial, and approved versions. Specifically, we employ a source code analysis tool namely Iterative Java Matcher (IJM) [173][4] to extract pairs of changed methods between the codebase and initial versions of a changed file, i.e., $\langle m_{\text{codebase}}, m_{\text{initial}} \rangle$; and pairs of changed methods between the initial and approved versions $\langle m_{\text{initial}}, m_{\text{approved}} \rangle$. Then, we merge each of $\langle m_{\text{codebase}}, m_{\text{initial}} \rangle$ and $\langle m_{\text{initial}}, m_{\text{approved}} \rangle$ into a triplet of the changed method $\langle m_{\text{codebase}}, m_{\text{initial}}, m_{\text{approved}} \rangle$ based on the whole content of m_{initial}. Note that in our approach, we also consider the newly-added m_{initial}, i.e., $\langle \emptyset, m_{\text{initial}}, m_{\text{approved}} \rangle$, but we do not consider deleted methods $\langle m_{\text{codebase}}, m_{\text{initial}}, \emptyset \rangle$ or $\langle m_{\text{codebase}}, \emptyset, \emptyset \rangle$ since our approach aims to transform the existing m_{initial} in a patch to the version that is reviewed and approved.

After we obtain a triplet of a changed method, we analyze the token-level difference information and integrate it into m_{initial}. We integrate the token-level code difference information into m_{initial} by inserting special tokens `<START_MOD>` and `<END_MOD>` to specify the code tokens that were changed from m_{codebase}. Figure 25 shows an example of m_{initial} with the token-level code difference information ($m_{\text{initial+diff}}$). To do so, we first identify the code tokens in m_{initial} that were changed from m_{codebase}. Specifically, we tokenize m_{initial} and m_{codebase} in sequences of tokens using the Java parser namely `javalang`[5]. Then, we use the approach of Liu *et al.* [174] to align code tokens between m_{initial} and m_{codebase} and identify the changed code tokens (i.e., either replaced or inserted). Finally, in m_{initial}, we insert `<START_MOD>` and `<END_MOD>` at the beginning and the end of each sequence of the changed tokens to produce $m_{\text{initial+diff}}$.

[4] https://github.com/VeitFrick/IJM
[5] https://github.com/c2nes/javalang

5.3.3 (Step 2) Training Phase

During the training phase, we leverage the CodeT5 pre-trained code model to learn code transformation in code review through the relationship between the pairs of $m_{\text{initial+diff}}$, m_{approved} in the training dataset. Prior to training a model, tokenization plays an important role to break source code into a meaningful sequence of code tokens. As suggested by prior work [14, 175, 176], we leverage the Byte-Pair Encoding (BPE) approach [151] to perform subword tokenization, i.e., split tokens into a list of sub-tokens. With the use of BPE subword tokenization, it will greatly reduce the vocabulary size, while enabling the model to create new tokens that never appear in the training dataset. We use the BPE tokenizer that is trained on the CodeSearchNet (CSN) corpus [177], provided by Wang *et al.* [26]. Since our approach uses special tokens (i.e., `<START_MOD>` and `<END_MOD>`) to explicitly specify which code tokens in the m_{initial} are changed from m_{codebase}, we include the special tokens in the BPE tokenizer to ensure that these tokens will not be split into sub-tokens. Then, we apply the BPE tokenizer to $m_{\text{initial+diff}}$ and m_{approved} to produce subtoken-level methods (i.e., $sm_{\text{initial+diff}}$ and sm_{approved}, respectively), where these subtoken-level methods contain a sequence of subtokens $sm = [\text{st}_1, ..., \text{st}_n]$. Finally, we feed these sequences into the CodeT5 pre-trained code models in order to learn the relationship between the input sequence of the initial version ($sm_{\text{initial+diff}}$) and the output sequence of the approved version (sm_{approved}) for each patch in the training dataset to build an NMT model. Below, we describe the technical details of the BPE tokenization and the architecture of CodeT5.

BPE Tokenization. The BPE tokenization process consists of two main steps: generating merge operations which determine how code tokens should be split, and applying merge operations to split code tokens into a list of sub-tokens. To generate merge operations, BPE first splits all code tokens into a list of characters. Next, BPE generates a merge operation by identifying the symbol pair (i.e., the pair of two consecutive characters) having the highest frequency in a corpus. Then, the co-occurrence characters appearing in the identified symbol pair are replaced by the symbol pair, without removing a single character that appears in the symbol pair. After that, the symbol pair is added to the vocabulary list. For example, suppose that ('a','c') has the highest frequency. The merge operation is ('a','c') $\rightarrow$ 'ac', which will replace all of the co-occurrence of ('a', 'c') with 'ac', without removing 'a' or 'c' that appears alone. The above steps are then repeated until a given number of merge operations is reached.

After the merge operations are generated, we apply the merge operations to split code tokens into a list of sub-tokens. To do so, BPE first splits all code tokens into sequences of characters. Then, the generated merge operations are applied to $m_{\text{initial+diff}}$ and m_{approved} in the training data, resulting $sm_{\text{initial+diff}}$ and sm_{approved}, respectively.

The Architecture of CodeT5. The architecture of the CodeT5 is based on the T5 architecture [170], which begins with a word embedding layer, followed by an encoder block and a decoder block. The architecture then ends with a linear layer and a softmax activation function.

Word Embedding Layer. Given $sm_{\text{initial+diff}}$ (or sm_{approved}) obtained from BPE tokenization, we generate an embedding vector for each sub-token and combine into an embedding matrix. Then, to capture the information related to the position of each token, unlike BERT [178] that combines embedding vectors with the vectors obtained from absolute position encoding, CodeT5 leverages relative position encoding to encode the position of tokens during self-attention calculation. The self-attention mechanism is described below.

The Encoder Block. The generated word embedding matrix of $sm_{\text{initial+diff}}$ is fed to the encoder block, which consists of six identical encoder layers. Similar to a Vanilla Transformer [105], each encoder layer is composed of two sub-components: a multi-head self-attention mechanism and a fully-connected feed-forward network (FFN). Before the input vector is fed to each sub-component, layer normalization [179] and residual skip connection [180] are respectively applied to the input vector. Different from Vanilla Transformer [105], the layer normalization here is a simplified version where additive bias is not applied.

Multi-head self-attention mechanism [105] is employed to allow CodeT5 to capture the contextual relationship between all tokens in the whole sequence. In particular, the multi-head self-attention mechanism calculates the attention score of each token by using the dot product operation. There are three main components for calculating the attention score: query (Q), key (K), and value (V) matrices. The attention weight is first computed by taking the scaled dot-product between Q and K. The softmax function is then applied to the result of the dot-product. Finally, the dot-product between the result obtained from the softmax function and V is calculated to update the value in V. However, different from Vanilla Transformer [105] that encodes the absolute position of each token before computing self-attention score, CodeT5 uses relative positional encoding to efficiently rep-

resent the position of tokens. Thus, an additional matrix P, which represents the relative position of tokens, is computed and supplied during self-attention calculation. Finally, the attention vectors obtained from the last encoder layer will be used to help decoder layers focus on appropriate tokens in the $sm_{\text{initial+diff}}$.

The Decoder Block. The generated embedding matrix of sm_{approved} is fed to the decoder block, which consists of six identical decoder layers. Each decoder layer is composed of the sub-components similar to the ones in the encoder layer. However, different from the encoder layer, the decoder layer uses special kinds of self-attention mechanism (i.e., masked multi-head self-attention and multi-head encoder-decoder self-attention) to generate attention vectors. The concept of the masked multi-head self-attention is that during the training phase, the attention of tokens that appear after the current tokens are masked out to ensure that the model does not attend to future (unseen) tokens while generating the next token. Then, the decoder layer uses the multi-head encoder-decoder self-attention mechanism to compute attention vectors, using the attention vectors generated by the encoder block and the matrix computed by the masked multi-head self-attention mechanism. After the decoder block computes attention vectors at the last layer, the linear layer will convert the attention vectors to the logits vector, having length equal to the vocabulary size. Then, the softmax function is applied to the logits vector to generate the vector, which determines the next token to be generated. Finally, the loss function will compare the generated sm_{approved} with the actual $sm_{approved}$ to estimate an error.

5.3.4 (Step 3) Inference Phase

After we fine-tune the CodeT5 pre-trained model, we aim to generate m_{approved} from the given $m_{\text{initial+diff}}$ in the testing dataset. To do so, we first tokenize $m_{\text{initial+diff}}$ to $sm_{\text{initial+diff}}$ by using BPE like in the training phase. Then, we leverage beam search to generate top-k best candidates of $m_{approved}$. The beam size k is a hyperparameter specifying the number of the best candidates to be generated. In particular, after a token is generated at each time step, the beam search uses a best-first search strategy to select the top-k candidates that have the highest conditional probability. The beam search will keep generating the top-k best candidates until the EOS token (i.e., "</s>") is generated.

5.4 Study Design

In this section, we provide the motivations of the research questions and describe our experimental setup.

5.4.1 Research Questions

We formulate the following two research questions in our study.

(RQ1) How do our approach and state-of-the-art code transformation approaches for code review perform under the time-wise evaluation scenario?
Motivation. Recently, Thongtanunam *et al.* [14] and Tufano *et al.* [24] proposed code transformation approaches for code review to facilitate developers during the code review process. However, as explained in Section 6.2, such approaches still have limitations that need to be addressed. Therefore, we formulate this RQ to investigate the accuracy of our proposed D-ACT approach and the existing code transformation approaches (i.e., AutoTransform [14] and TufanoT5 [24]) in the time-wise evaluation scenario.

(RQ2) What are the contributions of the components (i.e., the token-level code difference information and the pre-trained model) of our approach?
Motivation. Our D-ACT leverages the token-level code difference information and the pre-trained code model (i.e., CodeT5 [26]). However, little is known about the contribution of the token-level code difference and the pre-trained code model to our approach. Therefore, the effectiveness of our approach is possibly impacted when the token-level code difference is removed; or CodeT5 is changed to the other pre-trained programming language models or other code transformation approaches. Thus, we formulate this RQ to investigate the possible impact of the components of the token-level code difference and the pre-trained programming language models on the accuracy of our approach.

5.4.2 Experimental Setup

We now describe our data collection and data filtering approaches, the model implementation, the hyper-parameter settings in our experiments, and the evaluation measure.

Data Collection: In this work, we collect new datasets from the Gerrit code re-

Table 8: An overview statistic of the studied datasets.

Project	# Total patches	# Total studied patches	# Total studied triplets		
			# train	# validation	# test
Android	22,746	4,497 (19.77%)	14,690	1,836	1,835
Google	14,099	3,486 (24.73%)	9,899	1,237	1,235
Ovirt	22,800	5,802 (25.45%)	21,509	2,686	2,688

view repositories of Google, Ovirt, and Android which are software projects that code reviews are actively performed. We do not use the datasets of the prior studies [13, 14] since their datasets only have pairs of $\langle m_{\text{codebase}}, m_{\text{approved}} \rangle$, while our approach requires triplets of $\langle m_{\text{codebase}}, m_{\text{initial}}, m_{\text{approved}} \rangle$. To collect the datasets, we use Gerrit REST APIs to retrieve three versions of the submitted patches and their changed java files. Specifically, we obtain java files of the codebase and the initial version from the first version of a patch that was submitted for review. We then identify java files in the approved version that are changed from the initial version.

Data Filtering: We first follow the steps below to filter out method pairs ($\langle m_{\text{codebase}}, m_{\text{initial}} \rangle$ and $\langle m_{\text{initial}}, m_{\text{approved}} \rangle$) before forming triplets $\langle m_{\text{codebase}}, m_{\text{initial}}, m_{\text{approved}} \rangle$. Table 8 provides summary statistics of the dataset after data filtering steps are applied.

1. **Exclude patches that contain a single revision.** It is possible that the first version of the patches is approved without further revision. Thus, to ensure that our D-ACT can help developers to transform m_{initial} to m_{approved} that is different from m_{initial}, we exclude patches that contain a single revision.
2. **Exclude the method pairs having m_{initial} that appear more than once in the same file of a patch.** It is possible that developers revise different m_{codebase} to identical m_{initial}, or revise identical m_{initial} to different m_{approved}. Thus, there can be different method pairs containing the same m_{initial} but different m_{codebase} or m_{approved}. Consequently, when forming triplets by using such method pairs, there may be extra invalid triplets. Thus, to ensure the correctness of the triplets in the dataset, we exclude such method pairs.

After the method pairs are filtered out, they are used to form the triplets $\langle m_{\text{codebase}}, m_{\text{initial}}, m_{\text{approved}} \rangle$. The triplets are then filtered out according to the steps below.

1. **Exclude the triplets having m_{initial} that are the same as m_{codebase}.** During the code review process, developers tend to revise the modified or newly added methods rather than the whole files. However, there are some cases that m_{codebase} are the same as m_{initial}. Since the m_{initial} remain unchanged, we exclude such triplets.

2. **Exclude the triplets having m_{initial} that appear in more than one triplet.** Developers possibly revise identical m_{initial} to different m_{approved}, leading to triplets having the same m_{initial} but different m_{approved}. However, if an NMT model learns from these triplets, the NMT model may not learn how to correctly translate identical m_{initial} to different m_{approved}. Thus, similar to prior work [24], the triplets with the same m_{initial} but different m_{approved} are marked as duplicated and excluded from the dataset.

3. **Exclude the triplets having m_{initial} that are the same as m_{approved}.** Developers possibly do not revise m_{initial} until the submitted patch is approved, leading to triplets having m_{initial} the same as m_{approved}. However, if such triplets are included in the dataset, an NMT model may not learn how to generate m_{approved} that is different from m_{initial}. To ensure that the trained NMT model can generate the m_{approved} that is different from m_{initial}, we exclude such triplets.

4. **Exclude the triplets having m_{initial} or m_{approved} that contain more than 512 tokens** The size of methods can vary; however, an NMT model has a fixed input size (e.g., CodeBERT [155] can accept input that has at most 512 tokens). Similar to prior work [24], we exclude triplets having m_{initial} or m_{approved} that have more than 512 tokens.

Model Implementation: In the experiment, we implement our D-ACT, CodeBERT, GraphCodeBERT, PLBART, and CodeT5 by using the `HuggingFace` [181] library. In addition, we use the implementation of TufanoT5[6] that can be obtained from the `t5` library, which is implemented in Tensorflow. Similarly, we use the implementation of AutoTransform[7] that can be obtained from the `Tensor2Tensor` library, which is implemented in Tensorflow.

Hyper-parameter Setting: In our experiment, the hyper-parameter settings of the pre-trained models (i.e., CodeBERT, GraphCodeBERT, CodeT5, PLBART)

[6]https://github.com/RosaliaTufano/code_review_automation
[7]https://github.com/awsm-research/AutoTransform-Replication

are obtained from the base model (please refer to Huggingface website[8] for more detail). Similarly, for the baselines, we use the optimal hyper-parameter as reported in [14] (for AutoTransform) and [24] (for TufanoT5). During the model training phase, we use a batch size of 6, the maximum input length of 512, a learning rate of 5e-5, and the maximum training steps of 300,000. The validation loss is calculated every 2,000 training steps to indicate when to stop training. We use AdamW optimizer [182] to minimize training loss.

Evaluation Measure: Similar to the prior work [13, 14, 24], we measure *#perfect match* (PM) when evaluating D-ACT and the baselines. The PM is the number of generated m_{approved} that exactly matches the actual m_{approved} (the m_{approved} in ground-truth). To compare the generated m_{approved} and the actual m_{approved}, the generated m_{approved} and the actual m_{approved} are first tokenized by the Java parser obtained from the `javalang` library. Then, the sequence of tokens of the generated m_{approved} and the sequence of tokens of the actual m_{approved} are compared.

In our experiment, we use the beam size (k) of 1, 5 and 10 when generating m_{approved}. Thus, an NMT model will achieve a perfect match if one of the generated m_{approved} from the k sequence candidates matches the actual m_{approved}. We do not measure BLEU score [183] similar to the previous study [14], since Ding *et al.* [184] argue that two similar code sequences (i.e., two sequences having few different code tokens) may have different semantics. We also do not measure CodeBLEU [185] since CodeBLEU does not consider identifier names. Thus, the generated m_{approved} may have high CodeBLEU even though their identifier names are changed when compared to ground truth, indicating that m_{approved} are still incorrectly generated.

Experimental Environment: The experiment is conducted in the server equipped with the following hardware: an AMD Ryzen 9 5950X @3.4 GHz 16-Core CPU, an SSD of 1 Terabyte, a RAM of 64 Gigabyte, and an NVIDIA GeForce RTX 3090 GPU with 24 GB memory.

5.5 Results

In this section, we present the results with respect to our two research questions.

(RQ1) How do our approach and state-of-the-art code transformation approaches for code review perform under the time-wise evaluation scenario?

[8] https://huggingface.co

Table 9: (RQ1) #perfect match of our approach and the baselines (i.e., TufanoT5 and AutoTransform) for the time-wise evaluation scenario. The number in the parenthesis indicates the percentage improvement compared to TufanoT5 and AutoTransform, respectively.

	Google			Ovirt			Android		
Approach	k = 1	k = 5	k = 10	k = 1	k = 5	k = 10	k = 1	k = 5	k = 10
D-ACT	**74** (+362%, +2,367%)	**165** (+68%, +1,079%)	**202** (+62%, +1,022%)	**48** (+860%, +1,100%)	**187** (+307%, +523%)	**238** (+310%, +367%)	**12** (+1,100%, +1,200%)	**70** (+367%, +438%)	**101** (+261%, +274%)
TufanoT5	16	98	125	5	46	58	1	15	28
AutoTransform	3	14	18	4	30	51	0	13	27

Table 10: (RQ1) #perfect match of the existing code transformation approaches for the time-ignore and the time-wise evaluation scenarios.

		Google			Ovirt			Android		
Approach	Evaluation scenario	k = 1	k = 5	k = 10	k = 1	k = 5	k = 10	k = 1	k = 5	k = 10
TufanoT5	time-ignore	152	263	293	442	638	718	316	462	501
	time-wise	16	98	125	5	46	58	1	15	28
	% decrease	89.47	62.74	57.34	98.87	92.79	91.92	99.68	96.75	94.41
AutoTransform	time-ignore	114	208	233	578	794	854	344	489	516
	time-wise	3	14	18	4	30	51	0	13	27
	% decrease	97.37	93.27	92.27	99.31	96.22	94.03	100	97.34	94.77

Approach. To answer this RQ, we build and evaluate our approach and the baselines (i.e., TufanoT5 [24] and AutoTransform [14]) in the time-wise scenario. To do so, we first sort the triplets in chronological order by the commit date of their corresponding patches. Then, we split the triplets to train/validation/test set by the proportion of 80%/10%/10%, respectively. In this RQ, we also evaluate the baselines in the time-ignore scenario, where the triplets in the dataset are randomly shuffled.

To select a model for evaluation, we choose the model that achieves the lowest loss on the validation set. Finally, we measure PM (as explained in Section 5.4) on the test set. In particular, we compute the percentage increase of PM as

$$\frac{PM_{ours} - PM_{baseline}}{PM_{baseline}} \times 100\%$$

where PM_{ours} is the PM of our approach, and $PM_{baseline}$ is the PM of the baselines.

In addition, we measure PM of the time-ignore scenario ($PM_{time\text{-}ignore}$) and the time-wise scenario ($PM_{time\text{-}wise}$), achieved by the baselines. We compute the percentage decrease of PM as follows:

$$\frac{PM_{time\text{-}ignore} - PM_{time\text{-}wise}}{PM_{time\text{-}ignore}} \times 100\%$$

Result. **Our D-ACT can achieve PM at least 62% higher than TufanoT5 and at least 274% higher than AutoTransform.** Table 9 shows the results of PM of our D-ACT and the baselines across three datasets (i.e., Google, Ovirt and Android) based on the beam sizes of 1, 5 and 10. The table shows that for the beam size of 1, our D-ACT can achieve PM of 12 (Android) - 74 (Google), which is at least 362% higher than TufanoT5 and 1,100% higher than AutoTransform. On the other hand, for the beam size of 5, our D-ACT can achieve PM of 70 (Android) - 187 (Ovirt), which is at least 68.37% higher than TufanoT5 and 438.46% higher than AutoTransform. Likewise, we find that for the beam size of 10, our approach can achieve PM of 101 (Android) - 238 (Ovirt), which is at least 62% higher than TufanoT5 and 274% higher than AutoTransform. The results indicate that the number of m_{initial} that our D-ACT can correctly transform to m_{approved} is substantially higher than the TufanoT5 and AutoTransform approaches.

To investigate why our approach can outperform the TufanoT5 approach, we manually analyze the m_{approved} generated by our approach and the TufanoT5 approach (k = 1). To do so, we analyze the m_{approved} that only our approach can correctly generate while the TufanoT5 approach cannot.

We find that there are 65, 39, and 12 m_{approved} of Google, Ovirt, and Android, respectively that our approach can correctly generate while the TufanoT5 approach cannot. The results suggest that the higher PM of our approach has to do with the token-level code difference information and the CodeT5 [26] model of our approach.

When the existing code transformation approaches are evaluated in the time-wise scenario, PM of these approaches decreases by at least 57% (for TufanoT5) and 92% (for AutoTransform). Table 10 shows the PM of the existing code transformation approaches across the three datasets (i.e., Google, Ovirt and Android) in the time-ignore and the time-wise scenarios, based on the beam sizes of 1, 5 and 10. The table shows that when the beam size is 1, PM of TufanoT5 drops by at least 89.97% ($\frac{152-16}{152}$; for Google) and PM of AutoTransform drops by at least 97.37% ($\frac{114-3}{114}$; for Google). Even the beam size increases to 5 and 10, PM of TufanoT5 drops by at least from 57.34% ($\frac{293-125}{293}$; for Google at beam size of 10) to 62.74% ($\frac{263-98}{263}$; for Google at beam size of 5). Likewise,

PM of AutoTransform drops by at least from 91.92% ($\frac{718-58}{718}$; for Ovirt at beam size of 10) to 93.27% ($\frac{208-14}{208}$; for Google at beam size of 5). The results imply that PM of the existing code transformation approaches dramatically decreases when being evaluated in the time-wise evaluation scenario. Thus, future work should consider the time-wise scenario when evaluating proposed approaches.

Table 11: (RQ2) #perfect match of our approach and the CodeT5 model (here, the CodeT5 model is trained without the token-level code difference information) for the time-wise evaluation scenario; the number in the parenthesis indicates the percentage improvement compared to CodeT5.

	Google			Ovirt			Android		
Approach	k = 1	k = 5	k = 10	k = 1	k = 5	k = 10	k = 1	k = 5	k = 10
D-ACT	**74 (+1380%)**	**165 (+132%)**	**202 (+82%)**	**48 (+243%)**	**187 (+46%)**	**238 (+26%)**	**12 (+1200%)**	**70 (+35%)**	**101 (+17%)**
CodeT5	5	71	111	14	128	189	0	52	86

(RQ2) What are the contributions of the components (i.e., the token-level code difference information and the pre-trained model) of our approach? Approach. To answer this RQ, we investigate the changes of PM when the components of our D-ACT is varied. In particular, we extend our experiment by removing the token-level code difference information from our approach to examine the impact of the token-level code difference information on our approach. We also conduct experiment by changing CodeT5 to AutoTransform [14], TufanoT5 [24], and the following pre-trained programming language models: CodeBERT [155], GraphCodeBERT [156] and PLBART [157]. CodeBERT and GraphCodeBERT are the pre-trained encoder models based on BERT [178] architecture, while PLBART is the pre-trained encoder-decoder model based on BART [186] architecture.

Table 12: (RQ2) #perfect match of our approach and the variants of our approach, where CodeT5 is changed to other pre-trained PL models or other code transformation approaches (trained with the token-level code difference information) for the time-wise evaluation scenario.

	Google			Ovirt			Android		
Approach	k = 1	k = 5	k = 10	k = 1	k = 5	k = 10	k = 1	k = 5	k = 10
D-ACT	**74**	**165**	**202**	**48**	**187**	**238**	**12**	**70**	**101**
CodeBERT	48	76	78	15	49	64	1	10	18
GraphCodeBERT	45	83	92	17	46	62	3	19	28
PLBART	3	49	84	19	89	129	12	51	78
TufanoT5	16	88	122	16	85	113	2	33	55
AutoTransform	13	38	50	26	64	92	3	33	46

Similar to RQ1, we evaluate our approach and the variation of our approach in the time-wise scenario. We choose the model that achieves the lowest loss on the validation set, and measure PM on the test set. In addition, we calculate the percentage change of PM as

$$\frac{PM_{ours} - PM_{variants}}{PM_{ours}} \times 100\%$$

where $PM_{variants}$ is PM of the variation of our approach explained above, and PM_{ours} is PM of our approach.

Result. **Using the token-level code difference information can increase PM by 17%-82% when compared to without using the token-level code difference information.** Table 11 shows PM of our approach and the CodeT5 [26] model across the three datasets (i.e., Google, Ovirt and Android) based on the beam sizes of 1, 5 and 10. The table shows that

when the token-level code difference information is used together with CodeT5, at beam size of 1, PM increases by 243% ($\frac{48-14}{14}$; for Ovirt)-1,380% ($\frac{74-5}{5}$; for Google). Similarly, for the beam size of 5 and 10, PM increases by 35% ($\frac{70-52}{52}$; for Android) - 132% ($\frac{165-71}{71}$; for Google) and 17% ($\frac{101-86}{86}$; for Android) - 82% ($\frac{202-111}{111}$; for Google), respectively.

The results highlight the performance improvement (in term of PM) when incorporating the token-level code difference information to transform m_{initial} to m_{approved}.

Using CodeT5 can increase PM by at least 29% when compared to the other pre-trained code models.

Table 12 shows that when CodeT5 is used instead of CodeBERT [155], we find that PM increases by at least 54% ($\frac{74-48}{48}$; for Google), 117% ($\frac{165-76}{76}$; for Google), and 159% ($\frac{202-78}{78}$; for Google) for the beam size of 1, 5, and 10, respectively. Similarly, when CodeT5 is used instead of GraphCodeBERT [156], we find that PM increases by at least 64% ($\frac{74-45}{45}$; for Google), 99% ($\frac{165-83}{83}$; for Google), and ($\frac{202-92}{92}$; for Google) for the beam size of 1, 5, and 10, respectively. Lastly, when the CodeT5 is used instead of PLBART [157], we find that PM increases by at least 153% ($\frac{48-19}{19}$; for Ovirt), 37% ($\frac{70-51}{51}$; for Android), and 29% ($\frac{101-78}{78}$; for Android) for the beam size of 1, 5, and 10, respectively. The results indicate that

CodeT5 contributes to the performance improvement (in term of PM) over the other pre-trained code models when transforming $m_{\text{initial+diff}}$ to m_{approved}.

Using CodeT5 can increase PM by at least 85% and 66% when compared to the existing code transformation approaches for code review (i.e., AutoTransform [14] and TufanoT5 [24], respectively). Table 12 shows that when CodeT5 is used instead of TufanoT5, PM increases by at least 200% ($\frac{48-16}{16}$; for Ovirt), 88% ($\frac{165-88}{88}$; for Google) and 66% ($\frac{202-122}{122}$; for Google) for beam size of 1, 5, and 10, respectively. The similar observation can be seen when CodeT5 is used instead of AutoTransform. In particular, we find that PM increases by at least 85% ($\frac{48-26}{26}$; for Ovirt), 112% ($\frac{70-33}{33}$; for Android) and 120% ($\frac{101-46}{46}$; for Android) for beam size of 1, 5, and 10, respectively. The results indicate that CodeT5 contributes to the performance improvement (in term of PM) over the existing code transformation approaches when transforming $m_{\text{initial+diff}}$ to m_{approved}.

5.6 Discussion

5.6.1 Implications

Researchers should consider temporal information when evaluating their approaches for code review. The reason for this suggestion is that the experiment results of RQ1 show that PM of the existing code transformation approaches for code review decreases at least 57% (for TufanoT5 [24]) and 92% (for AutoTransform [14]). In reality, it is unlikely that the patches that are submitted later would be available for training a model, since patches are chronologically submitted to a code review platform. Thus, when conducting an experiment, patches should be sorted in chronological order before evaluating proposed approaches for code review.

5.6.2 The Performance of Our D-ACT on the Methods that Never Exist in a Codebase

In our studied dataset, there is a total of 2,139 m_{initial} in the test set that never exist in a codebase (764 for Android, 253 for Google, 1,122 for Ovirt). When we manually analyze the m_{initial} that are correctly transformed to m_{approved}, we find that our approach can correctly transform 2 out of 253 (for Google) and 11 out of 1,122 (for Ovirt) m_{initial} that never exist in a codebase to m_{approved}. The results imply that our approach is still far from perfect when transforming the newly added m_{initial} to m_{approved}. Thus, future work should develop an approach that can achieve higher accuracy for newly added methods.

5.6.3 The Model Complexity between D-ACT and TufanoT5

The model complexity plays a key role for practitioners when deciding to deploy a model in practice. Generally, highly complex models are likely to consume large memory, which requires premium GPU resources for deployment. There exist many studies [178, 187] showing that larger language models are likely to be more accurate whereas larger language models are likely to be more complex than others. Thus, we perform additional analysis to investigate the model complexity of our approach when compared to TufanoT5. To do so, we compute a total number of parameters of models obtained from the `parameters` function provided by the PyTorch library. We find that our D-ACT has approximately 222.8M parameters, while TufanoT5 has 60.5M parameters. While our model is more complex than TufanoT5, our model is still more accurate than TufanoT5. Nevertheless, our model can still fit within the commodity GPU (i.e., we used RTX 3090 in our experiment), which does not require any premium GPU resources like Nvidia DGX servers. Thus, our approach is still practical.

5.7 Threats to Validity

In this section, we disclose the threats to the validity of our study.

Threats to Construct Validity. The threats to construct validity relate to the data filtering process. As explained in Section 5.4, we perform data filtering to reduce the noise in our dataset (e.g., removing duplicated triplets). However, unknown noise may exist in our filtered dataset. Thus, future work should investigate our dataset to discover unknown noise, and explore its impact on code transformation approaches.

Threats to Internal Validity. The threats to internal validity relate to hyper-parameter settings that we use in our experiment. We do not explore all possible hyper-parameter settings when conducting the experiment. In addition, similar to Feng *et al.* [155], we use a constant learning rate instead of using warmup to adjust a learning rate during the training phase. Thus, the obtained results may differ from the reported results if hyper-parameter settings are changed or warmup is used. However, finding the best hyper-parameter setting for our approach is very expensive, and is not the primary goal of this paper. In contrast, the primary goal of our work is to fairly evaluate our approach and the baselines. To mitigate this threat, we provide detail of the hyper-parameter settings in the replication package to aid future replication studies.

Threats to External Validity. The threats to external validity relate to the generalizability of our approach. In our study, we evaluate our approach on three large-scale software projects on Gerrit (i.e., Google, Ovirt and Android). In addition, our study focuses on transforming methods in the patches written in Java. However, the results presented in our study may not be generalized to other software projects, other programming languages, or changes that occur outside methods. Thus, other software projects that are implemented in Java, other programming languages, or other changes that occur outside methods can be explored in future work.

5.8 Summary

In this chapter, we highlight the importance of integrating the token-level code difference information when designing a code transformation approach. Then, we proposed our D-ACT, which leverages the token-level code difference information, and CodeT5 [26]. In addition, the evaluation of the prior studies did not consider the chronological order of the data, which is not realistic. Therefore, we conducted the experiment by considering the chronological order of the data.

Through an empirical evaluation on three large-scale software projects on Gerrit (i.e., Google, Ovirt and Android), we found that (1) our D-ACT can achieve PM at least 62% higher than TufanoT5 [24] and at least 274% higher than AutoTransform [14]; (2) PM of the existing code transformation approaches decreases at least 57% (for TufanoT5) and 92% (for AutoTransform) when they are evaluated in the time-wise scenario; and (3) the token-level code difference information of our approach can increase PM by 17% - 82% when compared to without using the token-level code difference information.

6 Fine-Tuning and Prompt Engineering for Large Language Models-based Code Review Automation

Chapter Overview

The emergence of large language models (LLMs) has motivated researchers to leverage their capabilities for code review automation. Recent work focused on developing pre-trained models for code review automation, which requires expensive computing resources. Thus, prompt engineering is the common approach to leveraging LLMs for code review automation. While there exist different prompting techniques, previous work relating to prompt engineering for code review automation only relies on the code submitted for review without contextual information when constructing prompts for LLMs. Therefore, it is still difficult for practitioners to conclude which prompting technique is the best for code review automation.

In this chapter, to address the overarching research question of the thesis, we investigate the performance of LLMs when being prompted with the following prompting techniques: zero-shot learning, few-shot learning and persona. Zero-shot learning involves prompting LLMs to generate an output from a given instruction and an input. For the few-shot learning, we incorporate contextual information by including code from other patches that are similar to the current version of the submitted patch into prompts for LLMs. On the other hand, Persona involves prompting LLMs to act as a specific role or persona to ensure that LLMs will generate output that is similar to the output generated by a specified persona. Through the experimental study of the three code review automation datasets, we find that LLMs that are prompted by few-shot learning achieve the highest result.

The work in this chapter appears in Chanathip Pornprasit, and Chakkrit Tantithamthavorn. “Fine-tuning and prompt engineering for large language models-based code review automation.” Information and Software Technology (IST), 175, p.107523, 2024.

6.1 Introduction

Code review is a software quality assurance practice where developers other than an author (aka. reviewers) review a code change that the author creates to ensure the quality of the code change before being integrated into a codebase. While code review can ensure high software quality, code review is still time-consuming and expensive. Thus, neural machine translation (NMT)-based code review automation approaches were proposed [13, 14, 188] to facilitate and expedite the code review process. However, prior studies [17, 189] found that such approaches are still not perfect due to limited knowledge of the NMT-based code review automation models that are trained on a small code review dataset.

To address the aforementioned challenge of the NMT-based code review automation approaches, recent work proposed large language model (LLM)-based approaches for the code review automation task [17, 24]. A large language model is a large deep learning model that is based on the transformer architecture [105] and pre-trained on massive textual datasets. An example of the LLM-based code review automation approaches includes CodeReviewer [17], a pre-trained LLM that is based on the CodeT5 [26] model. Li *et al.* [17] showed that their proposed CodeReviewer outperforms prior NMT-based code review automation approaches [13, 14, 188]. However, the training process of CodeReviewer requires a lot of computing resources (i.e., two DGX-2 servers equipped with 32 NVIDIA V100 GPUs in total). Such large computing resources are infeasible for organizations with limited budgets.

Since pre-training LLMs for code review automation can be expensive, fine-tuning and prompt engineering are the two common approaches to leveraging LLMs for code review automation. In particular, fine-tuning involves further training LLMs that are already pre-trained on a specific code review dataset. For example, Lu *et al.* [190] proposed LLaMa-Reviewer, which is the LLM-based code review automation approach that is being fine-tuned on a base LLaMa model [191]. On the other hand, prompting [192–194] involves providing explicit instructions to guide the model's generation process without requiring a specific code review dataset. For instance, Guo *et al.* [16] conducted an empirical study to investigate the potential of GPT-3.5 for code review automation by using zero-shot learning with GPT-3.5.

While Guo *et al.* [16] demonstrate the potential of using GPT-3.5 for code review automation, their study still has the following limitations. First, the results

of Guo *et al.* [16] are limited to zero-shot GPT-3.5. However, there are other approaches to leverage GPT-3.5 (i.e., fine-tuning and few-shot learning) that are not included in their study. Thus, it is difficult for practitioners to conclude which approach is the best for leveraging LLMs for code review automation. Second, although prior studies [195–197] found that model fine-tuning can improve the performance of pre-trained LLMs, Guo *et al.* [16] did not evaluate the performance of LLMs when being fine-tuned. Thus, it is difficult for practitioners to conclude whether LLMs for code review automation should be fine-tuned to achieve the most effective results. Third, Guo *et al.* [16] did not investigate the impact of few-shot learning, which can improve the performance of LLMs over zero-shot learning [198–200]. Hence, it is difficult for practitioners to conclude which prompting strategy (i.e., zero-shot learning, few-shot learning, and a persona) is the most effective for code review automation.

In this work, we aim to investigate the performance of LLMs-based code review automation based on two contexts, i.e., when LLMs are leveraged by fine-tuning and prompting. In particular, we evaluate two LLMs (i.e., GPT-3.5 and Magicoder [201]) and the existing LLM-based code review automation approaches [17, 24, 189] with respect to the following evaluation measures: Exact Match (EM) [13, 189] and CodeBLEU [185]. Through the experimental study of the three code review automation datasets (i.e., $\text{CodeReviewer}_{\text{data}}$ [17], $\text{Tufano}_{\text{data}}$ [24], and $\text{D-ACT}_{\text{data}}$ [189]), we answer the following three research questions:

(RQ1) What is the most effective approach to leverage LLMs for code review automation?

Result. The fine-tuning of GPT 3.5 with zero-shot learning achieves 73.17% - 74.23% higher EM than the Guo *et al.*'s approach [16] (i.e., GPT 3-5 without fine-tuning). The results imply that GPT-3.5 should be fine-tuned to achieve the highest performance.

(RQ2) What is the benefit of model fine-tuning on GPT-3.5 for code review automation?

Result. The fine-tuning of GPT 3.5 with few-shot learning achieves 63.91% - 1,100% higher Exact Match than those that are not fine-tuned. The results indicate that fine-tuned GPT-3.5 can generate more correct revised code than GPT-3.5 without fine-tuning.

(RQ3) What is the benefit of few-shot learning on GPT-3.5 for code review automation?

Result. GPT-3.5 with few-shot learning achieves 46.38% - 659.09% higher Exact Match than GPT-3.5 with zero-shot learning. On the other hand, when a persona is included in input prompts, GPT-3.5 achieves 1.02% - 54.17% lower Exact Match than when the persona is not included in input prompts. The results indicate that the best prompting strategy when using GPT-3.5 without fine-tuning is using few-shot learning without a persona.

Recommendation. Based on our results, we recommend that (1) LLMs for code review automation should be fine-tuned to achieve the highest performance; and (2) when data is not sufficient for model fine-tuning (e.g., a cold-start problem), few-shot learning without a persona should be used for LLMs for code review automation.

Contributions. In summary, the main contributions of our work are as follows:

- We are the first to investigate the performance of LLMs-based code review automation when using model fine-tuning and inference techniques (i.e., zero-shot learning, few-shot learning and persona).
- We provide recommendations for adopting LLMs for code review automation to practitioners.

Open Science. Our fine-tuned models, script and results are made available online [202].

Paper Organization. Section 6.2 describes the related work and formulates research questions. Section 6.3 describes the study design of our study. Section 6.4 presents the experiment results. Section 6.5 discusses our experiment results. Section 6.6 describes possible threats to the validity. Section 6.7 draws the conclusions of our work.

6.2 Related Work and Research Questions

In this section, we provide the background knowledge of code review automation, discuss the existing large-language model-based code review automation approaches, and formulate the research questions.

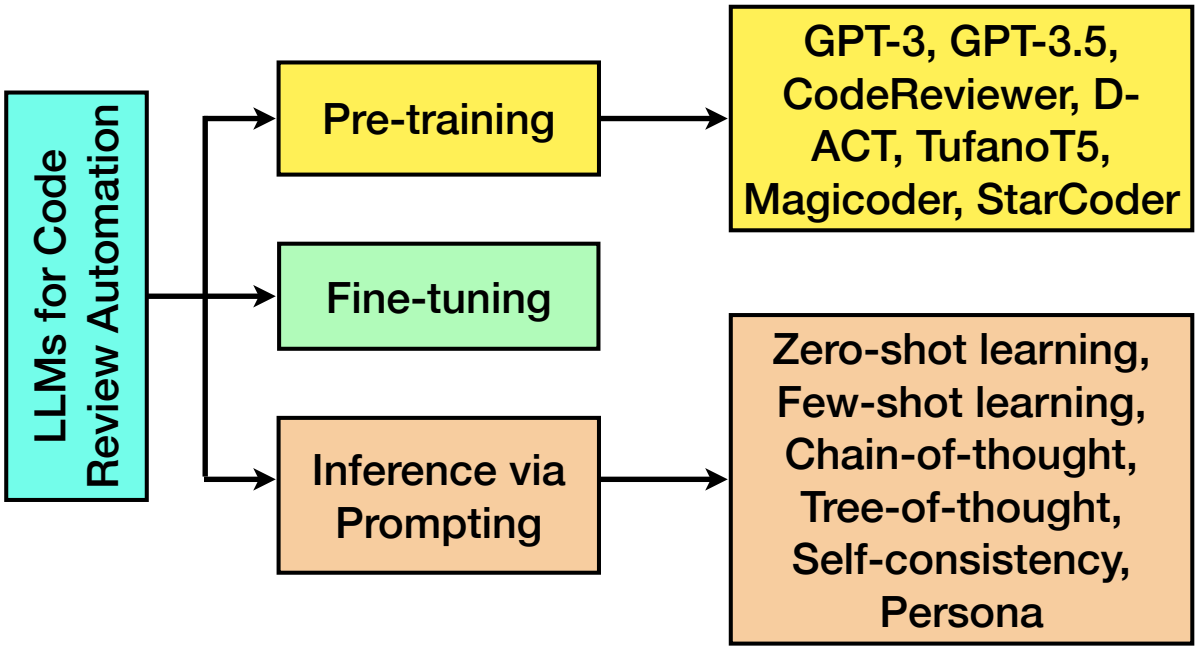


Figure 27: An overview of the modelling pipeline of LLMs for code review automation.

6.2.1 Code Review Automation

Code review is a software quality assurance practice where developers other than an author (aka. reviewers) provide feedback for a code change created by the author to ensure that the code change has sufficient quality to meet quality standards. While code review can ensure high software quality, code review is still time-consuming and expensive. Thus, developers still face challenges in receiving timely feedback from reviewers [5, 150]. Therefore, code review automation approaches [13, 15, 17, 24, 189] were proposed to facilitate and expedite the code review process.

Code review automation is generally formulated as a sequence generation task, where a language model is trained to learn the relationship between the submitted code and the revised code. Then, during the inference phase, the model aims to generate a revised version of a code change. Recently, neural machine translation (NMT)-based code review automation approaches were proposed [13, 14, 188]. Typically, NMT-based code review automation approaches are trained on a specific code review dataset. However, prior studies [17, 189] found that NMT-based code review automation approaches can perform well, but is still not per-

fect. This imperfect performance has to do with the limited knowledge of the NMT-based code review automation approaches that are being trained on a small code review dataset.

6.2.2 LLMs-based Code Review Automation Approaches

Large language models (LLMs) for code review automation refer to large language models specifically designed to support code review automation tasks, aiming to understand and generate source code written by developers and natural languages written by reviewers. Since source code and comments often have their own semantic and syntactical structures, recent work proposed various LLMs-based code review automation approaches [17, 24, 190]. For example, Li *et al.* [17] proposed CodeReviewer, a pre-trained LLM that is based on the CodeT5 model [26]. Prior studies found that LLMs-based code review automation approaches often outperform NMT-based ones [17, 24, 189]. For example, Li *et al.* [17] found that their proposed approach outperforms a transformer-based NMT model by 11.76%. Below, we briefly discuss the general modelling pipeline of LLMs for code review automation presented in Figure 27.

Model Pre-Training refers to the initial phase of training a large language model, where the model is exposed to a large amount of unlabeled data to learn general language representations. This phase aims to initialize the model's parameters and learn generic features that can be further fine-tuned for specific downstream tasks. Recently, there have been many large language models for code (i.e., LLMs that are specifically trained on source code and related natural languages). For example, the open-source community-developed large language models such as Code-LLaMa [203], StarCoder [204], and Magicoder [201]; and the commercial large language models such as GPT-3.5.

However, the development of large language models for code requires expensive GPU resources and budget. For example, GPT-3.5 requires 10,000 NVIDIA V-100 GPUs for model pre-training.[9] LLaMa2 [205] requires Meta's Research Super Cluster (RSC) as well as internal production clusters, which consists of approximately 2,000 NVIDIA A-100 GPUs in total. Therefore, many software organizations with limited resources and budgets may not be able to develop their large language models. Thus, fine-tuning and prompt engineering are the two common approaches to leverage the existing LLMs for code review automation

[9] https://gaming.lenovo.com/emea/threads/17314-The-hardware-behind-ChatGPT

when expensive GPU resources are not available for pre-training a large language model from scratch, where these techniques are more desirable for many organizations to quickly adopt new technologies.

Model Fine-Tuning is a common practice, particularly in transfer learning scenarios, where a model pre-trained on a large dataset (source domain, e.g., source code understanding) is adapted to a related but different task or dataset (target domain, e.g., code review automation). Recently, researchers have leveraged model fine-tuning techniques for LLMs to improve the performance of code review automation approaches. For example, Lu *et al.* [190] proposed LLaMa-Reviewer, which is an LLM-based code review automation approach that is being fine-tuned on a base LLaMa model [191] using three code review automation tasks, i.e., a review necessity prediction task to check if diff hunks need a review, a code review comment generation task to generates pertinent comments for a given code snippet, and a code refinement task to generate minor adjustment to the existing code. Lu *et al.* [190] found that the fine-tuning step on LLMs can greatly improve the performance of the existing code review automation approaches.

Inference refers to the process of using a pre-trained language model to generate source code based on a given natural language prompt instruction. Therefore, prompt engineering plays an important role in leveraging LLMs for code review automation to guide LLMs to generate the desired output. Different prompting strategies have been proposed.[10] For example, zero-shot learning, few-shot learning [198, 206, 207], chain-of-thought [208, 209], tree-of-thought [208, 209], self-consistency [210], and persona [194]. Nevertheless, not all prompting strategies are relevant to code review automation. For example, chain-of-thought, self-consistency and tree-of-thought promptings are not applicable to the code review automation task since they are designed for arithmetic and logical reasoning problems. Thus, we exclude them from our study.

In contrast, zero-shot learning, few-shot learning, and persona prompting are the instruction-based prompting strategies, which are more suitable for software engineering (including code review automation) tasks [211–214]. In particular, zero-shot learning involves prompting LLMs to generate an output from a given instruction and an input. On the other hand, few-shot learning [198, 206, 207] involves prompting LLMs to generate an output from N demonstration examples $\{(x_1, y_1), (x_2, y_2), ..., (x_N, y_N)\}$ and an actual input in a testing set, where x_i

[10] https://www.promptingguide.ai/

and y_i are the inputs and outputs obtained from a training set, respectively. Persona [194] involves prompting LLMs to act as a specific role or persona to ensure that LLMs will generate output that is similar to the output generated by a specified persona.

Table 13: The differences between our work and Guo *et al.*'s work [16].

	Guo *et al.* [16]	Our work
LLMs/approaches	GPT-3.5, CodeReviewer [17]	GPT-3.5, Magicoder [201], CodeReviewer [17], TufanoT5 [24], D-ACT [189]
Include fine-tuning LLMs?	No	Yes
Prompting techniques	Zero-shot learning, Persona	Zero-shot learning, Few-shot learning, Persona

6.2.3 GPT-3.5 for Code Review Automation

Recently, Guo *et al.* [16] conducted an empirical study to investigate the potential of GPT-3.5 for code review automation. However, their study still has the following limitations.

First, the results of Guo *et al.* [16] are limited to zero-shot GPT-3.5. In particular, Guo *et al.* [16] conducted experiments to find the best prompt for leveraging zero-shot learning with GPT-3.5. However, there are other approaches to leverage GPT-3.5 (i.e., fine-tuning and few-shot learning) that are not included in their study. The lack of a systematic evaluation of the use of fine-tuning and few-shot learning on GPT-3.5 makes it difficult for practitioners to conclude which approach is the best for leveraging LLMs for code review automation. To address this challenge, we formulate the following research question.

> RQ1: What is the most effective approach to leverage LLMs for code review automation?

Second, the performance of LLMs when being fine-tuned is still unknown. In particular, Guo *et al.* [16] did not evaluate the performance of LLMs when being fine-tuned. However, prior studies [195–197] found that model fine-tuning can improve the performance of pre-trained LLMs. The lack of experiments with model fine-tuning makes it difficult for practitioners to conclude whether LLMs for code review automation should be fine-tuned to achieve the most effective results. To address this challenge, we formulate the following research question.

RQ2: What is the benefit of model fine-tuning on GPT-3.5 for code review automation?

Third, the performance of LLMs for code review automation when using few-shot learning is still unknown. In particular, Guo *et al.* [16] did not investigate the impact of few-shot learning on LLMs for code review automation. However, recent work [198–200] found that few-shot learning could improve the performance of LLMs over zero-shot learning. The lack of experiments with few-shot learning on LLMs for code review automation makes it difficult for practitioners to conclude which prompting strategy (i.e., zero-shot learning, few-shot learning, and persona) is the most effective for code review automation. To address this challenge, we formulate the following research question.

RQ3: What is the benefit of few-shot learning on GPT-3.5 for code review automation?

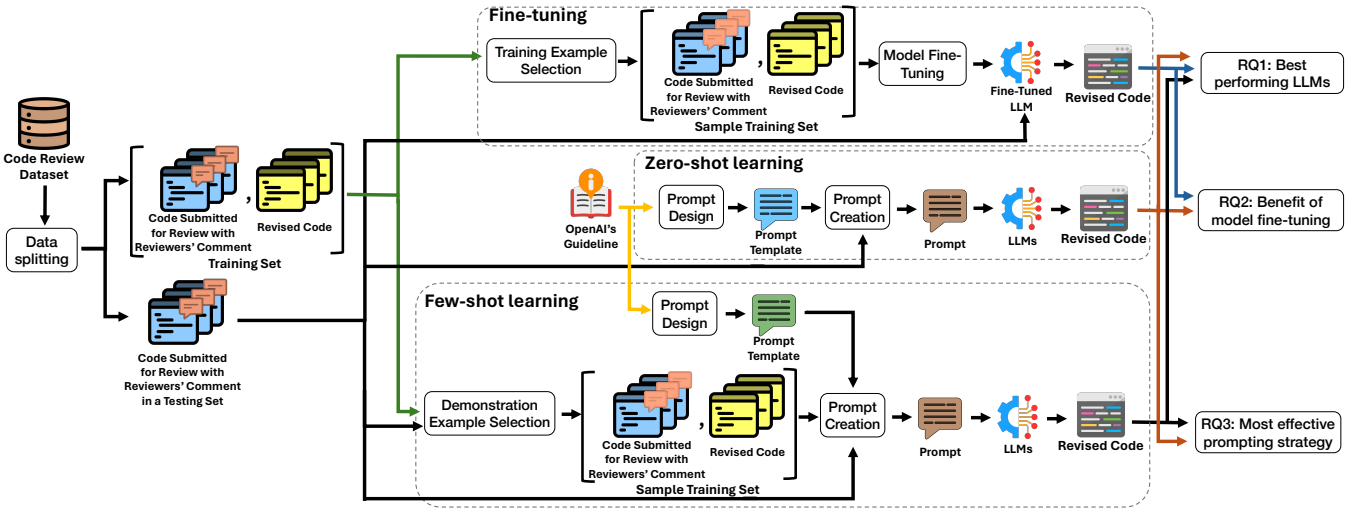


Figure 28: An overview of our experimental design (A persona is a part of zero-shot and few-shot learning).

6.3 Experimental Design

In this section, we provide an overview and details of our experimental design.

6.3.1 Overview

The goal of this work is to investigate which LLMs perform best when using model fine-tuning and inference techniques (i.e., zero-shot learning, few-shot learning [198, 206, 207], and persona [194]). To achieve this goal, we conduct

Table 14: Experimental settings in our study. We do not include experimental settings #3 and #4 since LLMs already learn the relationship between input (i.e., code submitted for review) and output (i.e., revised code).

Experimental setting	Fine-Tuning	Inference Technique	
		Prompting	Use Persona
#1	✓	Zero-shot	✗
#2			✓
#3		Few-shot	✗
#4			✓
#5	✗	Zero-shot	✗
#6			✓
#7		Few-shot	✗
#8			✓

experiments with two LLMs (i.e., GPT-3.5 and Magicoder [201]) on the following datasets that are widely studied in the code review automation literature [16, 189, 190, 215]: CodeReviewer$_{data}$ [17], Tufano$_{data}$ [24] and D-ACT$_{data}$ [189]. We use Magicoder [201] in our experiment since it is further trained on high-quality synthetic instructions and solutions.

In this study, we conduct experiments under six settings as presented in Table 14. According to the table, when the LLMs are fine-tuned, we use zero-shot learning with and without a persona. We do not use few-shot learning with the fine-tuned LLMs since the LLMs already learn the relationship between an input (i.e., code submitted for review) and an output (i.e., improved code). On the other hand, when the LLMs are not fine-tuned, we use zero-shot learning and few-shot learning, where each inference technique is used with and without a persona. Finally, we conduct 36 experiments in total (2 LLMs $\times$ 6 settings $\times$ 3 datasets).

Figure 28 provides an overview of our experimental design. To begin, the studied code review datasets are split into training and testing sets. The training set consists of the code submitted for review and reviewers' comments as input; and revised code as output. On the other hand, the testing set consists of only code submitted for review and reviewers' comments. Next, to fine-tune the studied LLMs, we first randomly obtain a set of training examples from the training set since using the whole training set is prohibitively expensive. Then, we use the selected training examples to fine-tune the studied LLMs. On the other hand, to use the inference techniques (i.e., zero-shot learning, few-shot learning

Table 15: A statistic of the studied datasets (the dataset of Android, Google and Ovirt are from the D-ACT$_{data}$ dataset [189]).

Dataset	# Train	# Validation	# Test	# Language	Granularity	Has Comment
CodeReviewer$_{data}$ [17]	150,405	13,102	13,104	9	Diff Hunk	✓
Tufano$_{data}$ [24]	134,238	16,779	16,779	1	Function	✓/✗
Android [189]	14,690	1,836	1,835	1	Function	✗
Google [189]	9,899	1,237	1,235	1	Function	✗
Ovirt [189]	21,509	2,686	2,688	1	Function	✗

and a persona), we first design prompt templates for each inference technique based on the guideline from OpenAI[11][12]. However, since few-shot learning requires demonstration examples, we select a set of demonstration examples for each testing sample from the training set. Then, we create prompts that look similar to the prompt templates. Finally, we use the studied LLMs to generate revised code from given prompts. We explain the details of the studied datasets, model fine-tuning, inference via prompting, evaluation measures, and hyper-parameter settings below.

6.3.2 The Studied Datasets

Recently, Tufano *et al.* [13, 188] collected datasets with the constraint that revised code must not contain the code tokens (e.g., identifiers) that do not appear in code submitted for review. Thus, such datasets do not align with the real code review practice since developers may add new code tokens when they revise their submitted code. Therefore, in this study, we use the CodeReviewer [17], TufanoT5 [24], and D-ACT [189] datasets, which do not have the above constraint in data collection instead. The details of the studied datasets are as follows (the statistic of the studied datasets is presented in Table 15).

- **CodeReviewer$_{data}$**: Li *et al.* [17] collected this dataset from the GitHub projects across nine programming languages (i.e., C, C++, C#, Java, Python, Ruby, php, Go, and Javascript). The dataset contains triplets of the code submitted for review (diff hunk granularity), a reviewer's comment, and the revised version of the code submitted for review (diff hunk granularity).
- **Tufano$_{data}$**: Tufano *et al.* [24] collected this dataset from Java projects in GitHub, and 6,388 Java projects hosted in Gerrit. Each record in the dataset

[11] https://help.openai.com/en/articles/6654000-best-practices-for-prompt-engineering-with-openai-api
[12] https://platform.openai.com/docs/guides/prompt-engineering/strategy-write-clear-instructions

contains a triplet of code submitted for review (function granularity), a reviewer's comment, and code after being revised (function granularity). Tufano *et al.* [24] created two types of this dataset (i.e., $\text{Tufano}_{\text{data}}$ (with comment) and $\text{Tufano}_{\text{data}}$ (without comment)).

- **D-ACT$_{\text{data}}$**: Pornprasit *et al.* [189] collected this dataset from the three Java projects hosted on Gerrit (i.e., Android, Google and Ovirt). Each record in the dataset contains a triplet of codebase (function granularity), code of the first version of a patch (function granularity), and code of the approved version of a patch (function granularity).

6.3.3 Model Fine-Tuning

To fine-tune the studied LLMs, as suggested by OpenAI, we first select a few training examples to fine-tune an LLM to see if the performance improves. Thus, we randomly select a set of examples from the whole training set by using the `random` function in Python to reduce bias in the data selection. However, there is no existing rule or principle to determine the number of examples that should be selected from a training set. Thus, we use the trial-and-error approach to determine the suitable number of training examples. To do so, we start by using approximately 6% training examples from the whole training set to fine-tune GPT-3.5. We find that GPT-3.5 that is fine-tuned with such training examples outperforms the existing code review automation approaches [17, 24, 189]. Therefore, based on the above finding, we use 6% training examples for the whole experiment.

After that, the selected training examples is used to fine-tune the studied LLMs. In particular, we fine-tune GPT-3.5 by using the API provided by OpenAI[13]. On the other hand, to fine-tune Magicoder [201], we leverage the state-of-the-art parameter-efficient fine-tuning technique called DoRA [216].

6.3.4 Inference via Prompting

In this work, we conduct experiments with the following prompting techniques: zero-shot learning, few-shot learning and a persona. We explain each prompting technique below.

For *zero-shot learning*, we first design the prompt template as presented in Fig-

[13]https://platform.openai.com/docs/guides/fine-tuning/create-a-fine-tuned-model

(Persona) You are an expert software developer in *<lang>*. You always want to improve your code to have higher quality.

(Instruction) Your task is to improve the given submitted code based on the given reviewer comment. Please only generate the improved code without your explanation.

(Input) *<input code>*
(Input) *<input comment>*

(a) A prompt template for zero-shot learning.

(Persona) You are an expert software developer in *<lang>*. You always want to improve your code to have higher quality. You have to generate an output that follows the given examples.

(Instruction and examples) You are given 3 examples. Each example begins with "##Example" and ends with "---". Each example contains the submitted code, the developer comment, and the improved code. The submitted code and improved code is written in *<lang>*. Your task is to improve your submitted code based on the comment that another developer gave you.

Example
Submitted code: *<code>*
Developer comment: *<comment>*
Improved code: *<code>*
--
<other examples>
--

(Input) Submitted code: *<input code>*
(Input) Developer comment: *<input comment>*

(b) A prompt template for few-shot learning.

Figure 29: Prompt templates for zero-shot learning and few-shot learning that contain simple instructions (*lang* refers to a programming language). The text in blue is omitted when reviewers' comments are not used in experiments.

ure 29a by following the guidelines from OpenAI[11][12] to ensure that the structure of the prompt is suitable for GPT-3.5. The prompt template consists of the following components: an instruction and an input (i.e., code submitted for review and a reviewer's comment).

Then, we create prompts by using the prompt template in Figure 29a and the code submitted for review with a reviewer's comment in a testing set. Finally, we use the LLMs to generate revised code from the created prompts.

For *few-shot learning* [198, 206, 207], we first design the prompt template as presented in Figure 29b. Similar to zero-shot learning, we follow the guidelines from OpenAI when designing the prompt template. The prompt template consists of the following components: demonstration examples, an instruction and an input (i.e., code submitted for review and a reviewer's comment).

In few-shot learning, demonstration examples are required to create a prompt. Thus, we select three demonstration examples, where each example consists of two inputs (i.e., code submitted for review and a reviewer's comment) and an output (i.e., revised code), by using BM25 [217]. We use BM25 [217] since prior work [193, 218] shows that BM25 [217] outperforms other sample selection approaches for software engineering tasks. In this work, we use BM25 [217] provided by the `gensim`[14] package. We select three demonstration examples for each testing sample since Gao *et al.* [192] showed that GPT-3.5 using three demonstration examples achieves comparable performance (i.e, 90% of the highest Exact Match) when compared to GPT-3.5 that achieves the highest performance by using 16 or more demonstration examples.

Then, we create prompts from the prompt template in Figure 29b; the code submitted for review and a reviewer's comment in the testing set; and the demonstration examples of the code submitted for review. Finally, we use LLMs to generate revised code from the prompts.

For *persona* [194], we include a persona in the prompt templates in Figure 29 to instruct GPT-3.5 to act as a software developer. We do so to ensure that the revised code generated by GPT-3.5 looks like the source code written by a software developer.

[14]https://github.com/piskvorky/gensim

Table 16: The evaluation results of GPT-3.5, Magicoder and the existing code review automation approaches.

Approach	Fine-Tuning	Inference Technique		CodeReviewer		Tufano (with comment)		Tufano (without comment)		Android		Google		Ovirt	
		Prompting	Use Persona	EM	CodeBLEU	EM	CodeBLEU	EM	CodeBLEU	EM	CodeBLEU	EM	CodeBLEU	EM	CodeBLEU
GPT-3.5	✓	Zero-shot	✗	37.93%	49.00%	22.16%	82.99%	6.02%	79.81%	2.34%	74.15%	6.71%	81.08%	3.05%	74.67%
			✓	37.70%	49.20%	21.98%	83.04%	6.04%	79.76%	2.29%	74.74%	6.14%	81.02%	2.64%	74.95%
	✗		✗	17.72%	44.17%	13.52%	78.36%	2.62%	74.92%	0.49%	61.85%	0.16%	61.04%	0.48%	56.55%
			✓	17.07%	43.11%	12.49%	77.32%	2.29%	73.21%	0.57%	55.88%	0.00%	50.65%	0.22%	45.73%
		Few-shot	✗	26.55%	47.50%	19.79%	81.47%	8.96%	79.21%	2.34%	75.33%	2.89%	81.40%	1.64%	73.83%
			✓	26.28%	47.43%	20.03%	81.61%	9.18%	78.98%	1.62%	74.65%	2.45%	81.07%	1.67%	73.29%
Magicoder	✓	Zero-shot	✗	27.43%	44.86%	11.14%	69.77%	1.97%	69.25%	0.27%	65.39%	0.57%	69.30%	0.30%	64.19%
			✓	27.98%	45.36%	11.06%	69.60%	2.12%	68.84%	0.65%	65.41%	1.13%	69.30%	1.00%	64.16%
	✗		✗	9.75%	39.45%	8.65%	73.90%	0.81%	59.49%	0.16%	47.37%	0.08%	48.82%	0.04%	44.38%
			✓	9.93%	39.48%	8.71%	73.57%	1.51%	67.65%	0.27%	47.46%	0.08%	48.58%	0.11%	43.65%
		Few-shot	✗	15.89%	36.24%	2.93%	4.36%	1.99%	7.49%	0.22%	37.61%	0.49%	42.50%	0.74%	40.33%
			✓	17.80%	38.93%	2.89%	3.70%	1.84%	6.96%	0.27%	16.59%	0.65%	18.83%	0.82%	19.55%
Guo *et al.* [16]	✗	Zero-shot	✓	21.77%	59.85%	-	-	-	-	-	-	-	-	-	-
CodeReviewer [17]	-	-	-	33.23%	55.43%	15.17%	80.83%	4.14%	78.76%	0.54%	75.24%	0.81%	80.10%	1.23%	75.32%
TufanoT5 [24]				11.90%	43.39%	14.26%	79.48%	5.40%	77.26%	0.27%	75.88%	1.37%	82.25%	0.19%	73.53%
D-ACT [189]				-	-	-	-	-	-	0.65%	75.99%	5.98%	81.85%	1.79%	79.77%

6.3.5 The Evaluation Measures

We use the following measures to evaluate the performance of the studied LLMs (i.e., GPT-3.5 and Magicoder [201]) and code review automation approaches (i.e., CodeReviewer [17], TufanoT5 [24], and D-ACT [189]).

1. **Exact Match (EM)** [17, 24, 189] is the number of the generated revised code that is the same as the actual revised code in the testing dataset. We use this measure since it is widely used for evaluating code review automation approaches [13, 24, 189]. To compare the generated revised code with the actual revised code, we first tokenize both revised code to sequences of tokens. Then, we compared the sequence of tokens of the generated revised code with the sequence of tokens of the actual revised code. A high value of EM indicates that a model can generate revised code that is the same as the actual revised code in the testing dataset.
2. **CodeBLEU** [185] is the extended version of BLEU (i.e., an n-gram overlap between the translation generated by a deep learning model and the translation in ground truth) [183] for automatic evaluation of the generated code. We do not measure BLEU like in prior work [17, 24] since Ren *et al.* [185] found that this measure ignores syntactic and semantic correctness of the generated code. In addition to BLEU, CodeBLEU considers the weighted n-gram match, matched syntactic information (i.e., abstract syntax tree: AST) and matched semantic information (i.e., data flow: DF) when computing the similarity between the generated revised code and the actual revised code. A high value of CodeBLEU indicates that a model can generate revised code that is syntactically and semantically similar to the actual revised code in the

testing dataset.

6.3.6 The Hyper-Parameter Settings

In this study, we use the following hyper-parameter settings when using GPT-3.5 to generate revised code: temperature of 0.0 (as suggested by Guo *et al.* [16]), top_p of 1.0 (default value), and max length of 512. To fine-tune GPT-3.5, we use hyper-parameters (e.g., number of epochs and learning rate) that are automatically provided by OpenAI API.

For Magicoder [201], we use the same hyper-parameter as GPT-3.5 to generate revised code. To fine-tune Magicoder, we use the following hyper-parameters for DoRA [216]: attention dimension (r) of 16, alpha (α) of 8, and dropout of 0.1 .

Table 17: The statistical details of GPT-3.5, Magicoder and the existing code review automation approaches.

Model	# parameters
GPT-3.5	175 B
Magicoder [201]	6.7 B
TufanoT5 [24]	60.5 M
CodeReviewer [17]	222.8 M
D-ACT [189]	222.8 M

6.4 Result

In this section, we present the results of the following three research questions.

(RQ1) What is the most effective approach to leverage LLMs for code review automation?

Approach. To address this RQ, we leverage fine-tuning and inference techniques (i.e., zero-shot learning, few-shot learning, and persona) on GPT-3.5 and Magicoder (The details of GPT-3.5 and Magicoder are presented in Table 17) . Then, we measure EM of the results obtained from GPT-3.5, Magicoder and Guo *et al.*'s approach [16].

Result. **The fine-tuning of GPT 3.5 with zero-shot learning helps GPT-3.5 to achieve 73.17% -74.23% higher EM than the Guo *et al.* [16]'s approach.** Table 16 shows the results of EM achieved by GPT-3.5, Magicoder and Guo *et*

```
public static void writeSegmentedCopyRatioPlot(final String sample_name, final
String tnFile, final String preTnFile, final String segFile, final String
outputDir, final Boolean log) {
-   String logArg = "FALSE";
-   if (log) {
-       logArg = "TRUE";
-   }
+   String logArg = log ? "TRUE" : "FALSE";
    final RScriptExecutor executor = new RScriptExecutor();
    executor.addScript(new Resource(R_SCRIPT,
CopyRatioSegmentedPlotter.class));
    executor.addArgs("--args", "--sample_name=" + sample_name, "--
targets_file=" + tnFile, "--pre_tn_file=" + preTnFile, "--seg_file=" +
segFile, "--output_dir=" + outputDir, "--log2_input=" + logArg);
    executor.exec();
}
```

(a) The difference between code submitted for review and revised code that GPT-3.5 with zero-shot learning (no persona) correctly generates.

```
- public static void writeSegmentedCopyRatioPlot(final String sample_name,
final String tnFile, final String preTnFile, final String segFile, final
String outputDir, final Boolean log) {
+ public static void writeSegmentedCopyRatioPlot(final String sampleName,
final String tnFile, final String preTnFile, final String segFile, final
String outputDir, final boolean log) {
-   String logArg = "FALSE";
-   if (log) {
-       logArg = "TRUE";
-   }
+ String logArg = log ? "TRUE" : "FALSE";
    final RScriptExecutor executor = new RScriptExecutor();
    executor.addScript(new Resource(R_SCRIPT, CopyRatioSegmentedPlotter.
class));
-   executor.addArgs("--args", "--sample_name=" + sampleName, "--
targets_file=" + tnFile, "--pre_tn_file=" + preTnFile, "--seg_file=" +
segFile, "--output_dir=" + outputDir, "--log2_input=" + logArg);
+   executor.addArgs("--args", "--sample_name=" + sample_name, "--
targets_file=" + tnFile, "--pre_tn_file=" + preTnFile, "--seg_file=" +
segFile, "--output_dir=" + outputDir, "--log2_input=" + logArg);
```

(b) The difference between code submitted for review and revised code that GPT-3.5 with zero-shot learning (use persona) incorrectly generates.

```
func (e *MessagingEngine) SubmitLocal(event interface{}) {
        e.unit.Launch(func() {
                err := e.process(e.me.NodeID(), event)
-               if engine.IsInvalidInputError(err) {
+               if err != nil {
                        e.log.Fatal().Err(err).Str("origin", e.me.NodeID().
String()).Msg("failed to submit local message")
                }
        })
```

(c) The difference between code submitted for review and revised code that GPT-3.5 with few-shot learning (no persona) correctly generates.

```
func(e * MessagingEngine) SubmitLocal(event interface {}) {
    e.unit.Launch(func() {
        err: = e.process(e.me.NodeID(), event)
+       if err != nil {
            if engine.IsInvalidInputError(err) {
                e.log.Fatal().Err(err).Str(""
                    origin "", e.me.NodeID().String()).Msg(""
                    failed to submit local message "")
+           } else {
+               e.log.Fatal().Err(err).Str("origin", e.me.NodeID().
String()).Msg("unexpected error while processing local message")
+           }
+       }
    })
+ }
```

(d) The difference between code submitted for review and revised code that GPT-3.5 with few-shot learning (use persona) incorrectly generates.

Figure 30: (RQ3) Examples of the difference between code submitted for review and revised code generated by GPT-3.5 with zero-shot learning and few-shot learning.

al.'s approach [16]. The table shows that when GPT-3.5 and Magicoder are fine-tuned, such models achieve 73.17% -74.23% and 26.00% - 28.53% higher EM than the Guo *et al.*'s approach [16], respectively.

The results indicate that model fine-tuning could help GPT-3.5 to achieve higher EM when compared to the Guo *et al.*'s approach [16]. The higher EM has to do with model fine-tuning. When GPT-3.5 or Magicoder is fine-tuned, such models learn the relationship between inputs (i.e., code submitted for review and a reviewer's comment) and an output (i.e., revised code) from a number of examples in a training set. On the contrary, Guo *et al.*'s approach [16] only relies on the instruction and given input to generate revised code, which GPT-3.5 never learned during model pre-training.

(RQ2) What is the benefit of model fine-tuning on GPT-3.5 for code review

automation?

Approach. To address this RQ, we fine-tune GPT-3.5 as explained in Section 6.3. Then, we measure EM and CodeBLEU of the results obtained from the fine-tuned GPT-3.5 and the non fine-tuned GPT-3.5 with zero-shot learning.

Result. **The fine-tuning of GPT 3.5 with zero-shot learning helps GPT-3.5 to achieve 63.91% - 1,100% higher EM than those that are not fine-tuned.** Table 16 shows that in terms of EM, the fine-tuning of GPT 3.5 with zero-shot learning helps GPT-3.5 to achieve 63.91% - 1,100% higher than those that are not fine-tuned. In terms of CodeBLEU, the fine-tuning of GPT 3.5 with zero-shot learning helps GPT-3.5 to achieve 5.91% - 63.9% higher than those that are not fine-tuned.

The results indicate that fine-tuned GPT-3.5 achieve higher EM and CodeBLEU than those that are not fine-tuned. During the model fine-tuning process, GPT-3.5 adapt to the code review automation task by directly learning the relationship between inputs (i.e., code submitted for review and a reviewer's comment) and an output (i.e., revised code) from a number of examples in a training set. In contrast, non fine-tuned GPT-3.5 is given only an instruction and inputs which are not presented during model pre-training. Therefore, fine-tuned GPT-3.5 can better adapt to the code review automation task than those that are not fine-tuned.

(RQ3) What is the benefit of few-shot learning on GPT-3.5 for code review automation?

Approach. To address this RQ, we use zero-shot learning and few-shot learning with *non fine-tuned* GPT-3.5, where each inference technique is used with and without a persona, to generate revised code as explained in Section 6.3. Then, similar to RQ2, we measure EM and CodeBLEU of the results obtained from GPT-3.5.

Result. **GPT-3.5 with few-shot learning achieves 46.38% - 659.09% higher EM than GPT-3.5 with zero-shot learning.** Table 16 shows that in terms of EM, the use of few-shot learning on GPT-3.5 helps GPT-3.5 to achieve 46.38% - 241.98% and 53.95% - 659.09% higher than the use of zero-shot learning on GPT-3.5 without a persona and with a persona, respectively. In terms of CodeBLEU, the use of few-shot learning on GPT-3.5 helps GPT-3.5 to achieve 3.97% - 33.36% and 5.55% - 60.27% higher than the use of zero-shot learning on GPT-

```
  } // Show the loading indicator until data has been
fetched.
- if (!totalPagesFromData && totalPagesFromData !== 0) {
+ if (totalPagesFromData === null) {
    return fullPageLoadingIndicator; }
…
```

(a) Example of the code change for *bug fixing* (modify if condition)

```
- protected synchronized void closeLedgerManagerFactory() {
+ protected void closeLedgerManagerFactory() {
    LedgerManagerFactory lmToClose;
    synchronized(this) {
...
```

(b) Example of the code change for *bug fixing* (remove `synchronized` keyword)

```
"userscripts", cmd)
     log.misc.debug("Userscript to run  {}".format(cmd_path))
-    runner.run(cmd, *args, env=env, verbose=verbose).
+    runner.run(cmd_path, *args, env=env, verbose=verbose)
     runner.finished.connect(commandrunner.deleteLater)
     runner.finished.connect(runner.deleteLater)
```

(c) Example of the code change for *refactoring* (change variable name)

```
public EventDefinition(IEventDeclaration declaration, Stream
InputReader streamInputReader) {
-    this.fDeclaration = declaration;
-    this.fStreamInputReader = streamInputReader;
+    fDeclaration = declaration;
+    fStreamInputReader = streamInputReader;
```

(d) Example of the code change for *refactoring* (remove `this` qualifier)

```
...
        return false;
            if (! ServerDB::serverExists(srvnum))
                return false;
-           if (! ServerDB::getConf(srvnum, “autostart”,Meta
::mp.bAutoStart).toBool())
-               return false;
            Server *s = new Server(srvnum, this);
            if (! s->bValid) {
                delete s;
...
```

(e) Example of the code change for *other* (remove if condition)

```
public void move() {
...
    if (!newX.equals(spriteBase.getX()) || !
newY.equals(spriteBase.getY())) {
        Logger.log(String.format("Monster moved from (%f,
%f) to (%f, %f)", spriteBase.getX(), spriteBase.getY(),
newX, newY));
    }
-   this.setChanged();
-   this.notifyObservers();
}
```

(f) Example of the code change for *other* (remove method call)

Figure 31: Example of the code changes of each type

3.5 without a persona and with a persona, respectively.

The results indicate that GPT-3.5 with few-shot learning can achieve higher EM and CodeBLEU than GPT-3.5 with zero-shot learning. When few-shot learning is used to generate revised code from GPT-3.5, GPT-3.5 has more information from given demonstration examples in a prompt to guide the generation of revised code from given code submitted for review and a reviewer’s comment. In other words, such demonstration examples could help GPT-3.5 to correctly generate revised code from a given code submitted for review and a reviewer’s comment.

When a persona is included in input prompts, GPT-3.5 achieves 1.02% - 54.17% lower EM than when the persona is not included in input prompts. Table 16 also shows that when a persona is included in prompts, GPT-3.5 with zero-shot and few-shot learning achieves 3.67% - 54.17% and 1.02% - 30.77% lower EM compared to when a persona is not included in prompts, respectively. Similarly, when a persona is included in prompts, GPT-3.5 with zero-shot and few-shot learning achieves 1.33% - 19.13% and 0.15% - 0.90% lower CodeBLEU compared to when a persona is not included in prompts, respectively.

The results indicate that when a persona is included in prompts, GPT-3.5 with

zero-shot and few-shot learning achieves lower EM and CodeBLEU. To illustrate the impact of the persona, we present an example of the revised code that GPT-3.5 generates when zero-shot learning with and without a persona is used, and an example of the revised code that GPT-3.5 generates when few-shot learning with and without a persona is used in Figure 30.

Figure 30a presents the revised code that GPT-3.5 with zero-shot learning (no persona) correctly generates. In this figure, GPT-3.5 suggests an alternative way to initialize variable `logArg`. In contrast, Figure 30b presents the revised code that GPT-3.5 with zero-shot learning (use persona) incorrectly generates. In this figure, GPT-3.5 suggests changing the variable type from `Boolean` to `boolean` and changing the variable name from `sample_name` to `sampleName` in addition to suggesting an alternative way to initialize variable `logArg`.

Figure 30c presents another example of the revised code that GPT-3.5 with few-shot learning (no persona) correctly generates. In this figure, GPT-3.5 suggests a new `if` condition. On the contrary, Figure 30d presents the revised code that GPT-3.5 with few-shot learning (use persona) incorrectly generates. In this figure, GPT-3.5 suggests an additional `if` statement and an additional `else` block.

The above examples imply that when a persona is included in prompts, GPT-3.5 tends to suggest additional incorrect changes to the submitted code compared to when a persona is not included in prompts.

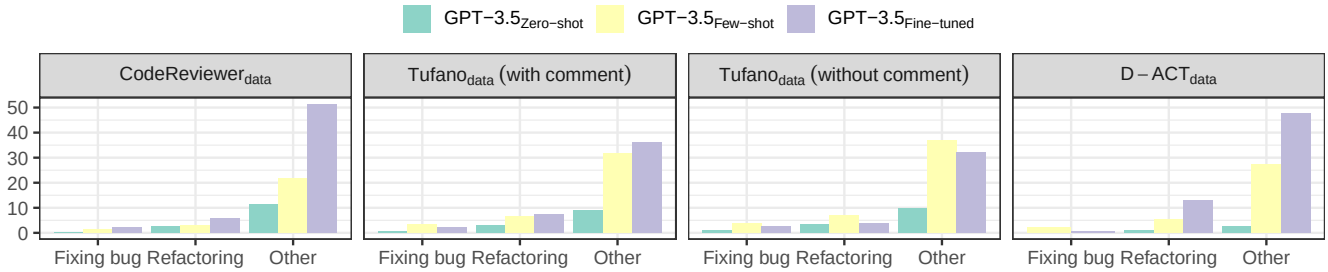
The above results indicate that the best prompting strategy when using GPT-3.5 without fine-tuning is few-shot learning without a persona.

6.5 Discussion

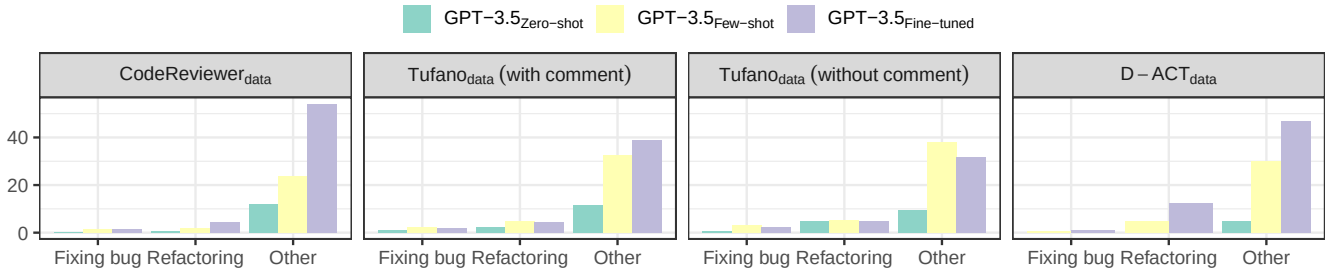
In this section, we discuss the implications of our findings, the additional results of GPT-3.5, and the cost and benefits of using GPT-3.5.

6.5.1 Implications of Our Findings

GPT-3.5 does not require a lot of training data for model fine-tuning to adapt to the code review automation task since Table 16 shows that GPT-3.5 that is fine-tuned on a subset of a training set outperforms the studied code review automation approaches [17, 24, 189]. The results imply that GPT-3.5 can adapt to the code review automation task by learning from a small set of training exam-



(a) The results of GPT-3.5 (persona is included in prompts)



(b) The results of GPT-3.5 (persona is not included in prompts)

Figure 32: The EM achieved by GPT-3.5$_{\text{Zero-shot}}$, GPT-3.5$_{\text{Few-shot}}$ and GPT-3.5$_{\text{Fine-tuned}}$ categorized by the types of code change. Here, GPT-3.5$_{\text{Zero-shot}}$ and GPT-3.5$_{\text{Few-shot}}$ refer to non fine-tuned GPT-3.5 with zero-shot learning and few-shot learning, respectively. On the other hand, GPT-3.5$_{\text{Fine-tuned}}$ refers to fine-tuned GPT-3.5 with zero-shot learning.

ples (approximately 20k training examples in this study), unlike the studied code review automation approaches that require the whole training set to adapt to the code review automation task.

Recommendations to practitioners. LLMs for code review automation should be fine-tuned to achieve the highest performance. The reason for this recommendation is the results of RQ2 show that fine-tuned GPT-3.5 outperforms those that are not fine-tuned. In contrast, when data is not sufficient for model fine-tuning (e.g., a cold-start problem), few-shot learning without a persona should be used for LLMs for code review automation. The reason for this recommendation is the results of RQ3 show that GPT-3.5 with few-shot learning outperforms GPT-3.5 with zero-shot learning, and GPT-3.5 without a persona outperforms GPT-3.5 with persona.

6.5.2 The Characteristics of the Revised Code that are Correctly Generated by GPT-3.5

The results of RQ2 and RQ3 demonstrate the benefits of model fine-tuning and few-shot learning on GPT-3.5 for the code review automation task, respectively. However, practitioners still do not clearly understand the characteristics of the

code changes of the revised code that GPT-3.5 correctly generates. To address this challenge, we aim to qualitatively investigate the revised code that GPT-3.5 can correctly generate. To do so, we randomly select the revised code that is only correctly generated by a particular model (e.g., we randomly obtain the revised code that only fine-tuned GPT-3.5 correctly generates while the others do not.) by using a confidence level of 95% and a confidence interval of 5%. Then, we classify the code changes of the selected revised code into the following categories based on the taxonomy of code change created by Tufano *et al.* [13] (the examples of the code changes in each category are depicted in Figure 31):

- *fixing bug*: The code changes in this category involve fixing bugs in the past. The following sub-categories are related to this category: exception handling, conditional statement, lock mechanism, method return value, and method invocation.
- *refactoring*: The code changes in this category involve making changes to code structure without changing the behavior of the changed code. The following sub-categories are related to this category: inheritance; encapsulation; methods interaction; readability; and renaming parameter, method and variable.
- *other*: The code changes that cannot be classified as neither *fixing bug* nor *refactoring* will fall into this category.

Figure 32 presents the characteristics of the code changes (i.e., *Fixing Bug*, *Refactoring* and *Other*) of the revised code that GPT-3.5 correctly generates. According to the figure, for the $\text{Tufano}_{\text{data}}$ (with comment), $\text{CodeReviewer}_{\text{data}}$ and $\text{D-ACT}_{\text{data}}$ dataset, the fine-tuning of GPT 3.5 with zero-shot learning helps GPT-3.5 achieves the highest EM for the code changes of *Refactoring* and *Other*. In contrast, for the $\text{Tufano}_{\text{data}}$ (without comment) dataset, non fine-tuned GPT-3.5 with few-shot learning achieves the highest EM for the code changes of all categories. According to Figure 32a and Figure 32b, we also find that fine-tuned GPT-3.5 and non fine-tuned GPT-3.5 that zero-shot and few-shot learning are used achieve the highest EM for the code changes of type *other* across all studied datasets. The reason for this result is that we do not specify the characteristics of the revised code (i.e., *fixing bug* and *refactoring*) in prompts. Thus, such models possibly generate revised code that is not specific to *fixing bug* or *refactoring*. Therefore, the majority of the code changes of the generated revised code are categorized as *other*.

Table 18: The evaluation results of GPT-3.5 when being fine-tuned with different sizes of training sets.

	CodeReviewer		Tufano (with comment)		Tufano (without comment)		Android		Google		Ovirt	
Size of Training Set	EM	CodeBLEU	EM	CodeBLEU	EM	CodeBLEU	EM	CodeBLEU	EM	CodeBLEU	EM	CodeBLEU
6%	37.93%	49.00%	22.16%	82.99%	**6.02%**	79.81%	2.34%	74.15%	6.71%	81.08%	**3.05%**	74.67%
10%	37.72%	48.83%	22.31%	83.43%	5.37%	**80.68%**	**2.51%**	75.10%	5.98%	80.65%	2.71%	75.13%
20%	**38.80%**	**49.33%**	**22.84%**	**83.44%**	5.65%	80.42%	2.34%	**76.04%**	**7.52%**	**81.40%**	2.83%	**75.46%**

6.5.3 The Impact of the Size of Training Dataset on Fine-Tuned GPT-3.5

The results of RQ2 show that model fine-tuning can help increase the performance of GPT-3.5. However, little is known whether fine-tuned GPT-3.5 can achieve higher performance when being fine-tuned with larger training sets. Therefore, we conduct experiments by using 10% and 20% of training sets to fine-tune GPT-3.5 (we do not use persona in these experiments).

Table 18 shows the results of EM and CodeBLEU that fine-tuned GPT-3.5 achieves across different sizes of training sets. The table shows that GPT-3.5 that is fine-tuned with 20% of a training set achieves 2.29% - 12.07% higher EM and 0.54% - 2.55% higher CodeBLEU than GPT-3.5 that is fine-tuned with 6% of a training set. In addition, GPT-3.5 that is fine-tuned with 20% of a training set achieves 2.38% - 25.75% higher EM and 0.01% - 1.25% higher CodeBLEU than GPT-3.5 that is fine-tuned with 10% of a training set, respectively. The results indicate that fine-tuned GPT-3.5 achieves higher performance when being fine-tuned with a larger training set.

Table 19: The evaluation results of GPT-3.5 for different prompt templates. P1 refers to the prompt templates with simple instructions (Figure 29). P2 refers to the prompt templates with instructions being broken down into smaller steps.(Figure 33). P3 refers to the prompt templates with detailed instructions (Figure 34).

		CodeReviewer		Tufano (with comment)		Tufano (without comment)		Android		Google		Ovirt	
Prompt Design	Prompting	EM	CodeBLEU	EM	CodeBLEU	EM	CodeBLEU	EM	CodeBLEU	EM	CodeBLEU	EM	CodeBLEU
P1	Zero-shot	17.72%	44.17%	13.52%	78.36%	2.62%	74.92%	0.49%	61.85%	0.16%	61.04%	0.48%	56.55%
P2		14.47%	43.52%	11.24%	79.05%	2.25%	76.54%	0.54%	66.10%	0.16%	67.07%	0.33%	60.76%
P3		11.94%	41.18%	9.86%	76.18%	1.26%	72.31%	0.05%	53.03%	0.08%	46.22%	0.26%	41.20%
P1	Few-shot	26.55%	47.50%	19.79%	81.47%	8.96%	79.21%	2.34%	75.33%	2.89%	81.40%	1.64%	73.83%
P2		25.25%	48.45%	15.82%	80.16%	6.84%	76.50%	0.60%	75.94%	3.56%	81.40%	1.67%	74.18%
P3		25.14%	48.60%	15.07%	79.83%	5.81%	75.76%	0.38%	74.95%	2.91%	81.18%	1.49%	73.31%

6.5.4 The Impact of Prompt Design on GPT-3.5

In RQ3, we use the prompt templates in Figure 29 that contain simple instructions to conduct experiments. However, prior work [219, 220] found that the design of prompts has an impact on the performance of LLMs. Thus, we further investigate

(Instruction) Follow the steps below to improve the given submitted code
step 1 - read the given submitted code and a reviewer comment
step 2 - identify lines that need to be modified, added or deleted
step 3 - generate the improved code without your explanation.

(Input) *<input code>*
(Input) *<input comment>*

(a) A prompt template for zero-shot learning.

(Instruction and examples) You are given 3 examples. Each example begins with "##Example" and ends with "---". Each example contains the submitted code, the developer comment, and the improved code. The submitted code and improved code is written in *<lang>*.

Follow the steps below to improve the given submitted code
step 1 - read the given submitted code and a reviewer comment in the above examples
step 2 - identify lines that need to be modified, added or deleted in the examples
step 3 - read the given submitted code and a reviewer comment
step 4 - identify lines that need to be modified, added or deleted
step 5 - generate the improved code without your explanation.

Example
Submitted code: *<code>*
Developer comment: *<comment>*
Improved code: *<code>*
--
<other examples>
--

(Input) Submitted code: *<input code>*
(Input) Developer comment: *<input comment>*

(b) A prompt template for few-shot learning.

Figure 33: Prompt templates for zero-shot learning and few-shot learning that the instructions are broken into smaller tasks (*lang* refers to a programming language). The text in blue is omitted when reviewers' comments are not used in experiments.

(Instruction) A developer asks you to help him improve his submitted code based on the given reviewer comment. He emphasizes that the improved code must have higher quality, conforms to coding convention or standard, and works correctly. He tells you to refrain from putting the submitted code in a class or method, and providing global variables or an implementation of methods that appear in the submitted code. He asks you to recommend the improved code without your explanation.

(Input) *<input code>*
(Input) *<input comment>*

(a) A prompt template for zero-shot learning.

(Instruction and examples) You are given 3 examples. Each example begins with "##Example" and ends with "---". Each example contains the submitted code, the developer comment, and the improved code. The submitted code and improved code is written in *<lang>*.

A developer asks you to help him improve his submitted code based on the given reviewer comment. He emphasizes that the improved code must have higher quality, conforms to coding convention or standard, and works correctly. He tells you to refrain from putting the submitted code in a class or method, and providing global variables or an implementation of methods that appear in the submitted code. He asks you to recommend the improved code without your explanation.

Example
Submitted code: *<code>*
Developer comment: *<comment>*
Improved code: *<code>*
--
<other examples>
--

(Input) Submitted code: *<input code>*
(Input) Developer comment: *<input comment>*

(b) A prompt template for few-shot learning.

Figure 34: Prompt templates for zero-shot learning and few-shot learning that more details are added to the instructions (*lang* refers to a programming language). The text in blue is omitted when reviewers' comments are not used in experiments.

the impact of prompt design on GPT-3.5 for code review automation. To do so, we conduct experiments by using the following two new prompt designs (we do not include a persona in these prompt designs since the results of RQ3 show that GPT-3.5 without a persona outperforms GPT-3.5 with a persona). First, we use the prompt design that a single instruction is broken into smaller steps[15], as depicted in Figure 33. Second, we use the prompt design that contains more detailed instructions[16], as depicted in Figure 34.

Table 19 shows the results of EM and CodeBLEU that GPT-3.5 achieves across different prompt designs. The table shows that for zero-shot learning, GPT-3.5 that is prompted by the prompt with a simple instruction achieves 16.44% - 45.45% higher EM than GPT-3.5 that is prompted by the prompt with an instruction being broken down into smaller steps. In addition, GPT-3.5 that is prompted by the prompt with a simple instruction achieves 37.12% - 880.00% higher EM than GPT-3.5 that is prompted by the prompt with a detailed instruction.

The table also shows that for few-shot learning, GPT-3.5 that is prompted by the prompt with a simple instruction achieves 5.15% - 290.00% higher EM than GPT-3.5 that is prompted by the prompt with an instruction being broken down into smaller steps. Furthermore, GPT-3.5 that is prompted by the prompt with a simple instruction achieves 5.61% - 515.79% higher EM than GPT-3.5 that is prompted by the prompt with detailed instruction.

The results indicate that GPT-3.5 that is prompted by the prompts with a simple instruction achieves the highest EM when compared to other prompt designs. Thus, the results imply that the prompt with a simple instruction is the most suitable for GPT-3.5 for code review automation.

6.5.5 Cost and Benefits of Using GPT-3.5 for Code Review Automation

Cost is one of the factors that determine the choices of AI services for practitioners. In the case of GPT-3.5 provided by OpenAI, the cost of using GPT-3.5 varies depending on usage[17]. In particular, the cost of using zero-shot learning and few-shot learning with GPT-3.5 is approximately 0.002 USD per query (for 1k input tokens and 1k output tokens) and 0.0035 USD per query (for 4k input tokens and 1k output tokens), respectively. On the other hand, the cost for fine-tuning

[15]https://learn.microsoft.com/en-us/azure/ai-services/openai/concepts/advanced-prompt-engineering?pivots=programming-language-chat-completions#break-the-task-down
[16]https://www.promptingguide.ai/introduction/tips
[17]https://openai.com/pricing

one GPT-3.5 is approximately 40 USD (we use approximately 8k examples from a training set), and the cost for using fine-tuned GPT-3.5 is approximately 0.009 USD per query (for 1k input tokens and 1k output tokens).

Assume that a software developer uses GPT-3.5 to help him/her review code submitted for review 1,000 times per day on average and he/she works for 25 days per month, the total GPT-3.5 usage is 1,000 $\times$ 25 $\times$ 12 = 300,000 times per year. Such GPT-3.5 usage accounts for $600 per year (300,000 $\times$ 0.002) when using zero-shot learning with GPT-3.5, $1,050 per year (300,000 $\times$ 0.0035) when using few-shot learning with GPT-3.5, and $2,740 per year (40 + (300,000 $\times$ 0.009)) when using fine-tuned GPT-3.5. However, when compared to the average yearly salary of software engineers around the world[18], the usage cost of GPT-3.5 is approximately 62.23% - 91.73% and 97.51% - 99.46% less than the lowest average salary (i.e., $7,255 in Nigeria) and the highest average salary (i.e., $110,140 in the United States), respectively. Nevertheless, the results of RQ1 show that fine-tuned GPT-3.5 achieves the highest EM. In addition, the results of RQ3 show that the use of few-shot learning without persona on GPT-3.5 helps GPT-3.5 achieve the highest EM and CodeBLEU. Thus, we recommend that GPT-3.5 for code review automation should be fine-tuned. Otherwise, leveraging GPT-3.5 by using few-shot learning without persona can be considered as an alternative.

6.6 Threats to Validity

We describe the threats to the validity of our study below.

6.6.1 Threats to Construct Validity

Threats to construct validity relate to the example selection techniques that we use to select examples for few-shot learning, and the design choices of the persona. We explain each threat below.

In this study, we only use the BM25 [217] technique to select three demonstration examples for few-shot learning. However, using more demonstration examples or different techniques to select demonstration examples may lead to results that differ from the reported results. Thus, more demonstration examples and other techniques for selecting demonstration examples can be explored in future work.

[18] https://codesubmit.io/blog/software-engineer-salary-by-country/

Since the code review automation task is related to revising the patches written by software developers, we use the persona (i.e., *an expert software developer*) to ensure that the revised code generated by GPT-3.5 looks like the source code written by a software developer. However, there are other similar personas (e.g., a senior software engineer, or a front-end software developer) that we do not explore. Thus, future work can explore other design choices of prompt and persona to find the optimal prompt and persona for code review automation tasks.

6.6.2 Threats to Internal Validity

Threats to internal validity relate to the randomness of GPT-3.5 and Magicoder, and the hyper-parameter settings that we use to fine-tune GPT-3.5 and Magicoder. The results that we obtain from GPT-3.5 and Magicoder may vary due to the randomness of GPT-3.5 and Magicoder. However, doing the same experiments multiple rounds can be expensive due to large testing datasets. Finally, we do not explore all possible combinations of hyper-parameter settings (e.g., the number of epoch or learning rate) when fine-tuning GPT-3.5 and Magicoder. We do not do so since the search space of hyper-parameter settings is large, which can be expensive. Nonetheless, the main goal of this study is not to find the best hyper-parameter settings for code review automation, but to investigate the performance of GPT-3.5 and Magicoder on code review automation tasks when using model fine-tuning.

6.6.3 Threats to External Validity

Threats to external validity relate to the generalizability of our findings in other software projects. In this study, we conduct the experiment with the dataset obtained from recent work [17, 24, 189]. However, the results of our experiment may not be generalized to other software projects. Thus, other software projects can be explored in future work.

Another threat relates to the updates to GPT-3.5 made by OpenAI in future. Due to the updates, reproduced experiment results may differ from those reported in this paper.

6.7 Summary

In this chapter, we investigate the performance of LLMs (i.e., GPT-3.5) for code review automation when using and inference techniques (i.e., zero-shot learning,

few-shot learning, and persona). We also compare the performance of the LLMs with the existing code review automation approaches [17, 24, 189]. Our results show that the best prompting strategy when using GPT-3.5 without fine-tuning is few-shot learning without a persona. Based on the results, we recommend that when data is not sufficient for model fine-tuning, few-shot learning without a persona should be used for LLMs for code review automation.

7 Conclusion

In this thesis, we identify the following limitations of AI-powered code review assistants due to a lack of contextual information: (1) defect prediction models lack contextual information in the form of the patches that are similar to the predicted patches, or code tokens from surrounding lines, to generate the explanation of their predictions, and (2) approaches for code review automation lack contextual information in the form of previous versions of the submitted patch or other similar patches to recommend the improved version of the submitted patch. These limitations lead to the hypothesis that AI-powered code review assistants should not solely rely on the technical information, but also on contextual information of its technical information. To validate the hypothesis, we formulate the following research question: how do we integrate contextual information into AI-powered code review assistants?

To address this RQ, we have the following contributions:

(1) We present PyExplainer, a local rule-based model-agnostic technique for explaining the predictions of machine learning-based JIT defect prediction models (Chapter 2).

(2) We present JITLine, a line-level defect prediction model to identify defective lines in a defective patch (Chapter 3).

(3) We present DeepLineDP, a line-level defect prediction model to identify defective lines in a defective file (Chapter 4).

(4) We present D-ACT, a Neural Machine Translation-based code transformation approach to transform the first version of a patch to the version that reviewers are likely to approve(Chapter 5).

(5) We investigate the performance of LLMs-based code review automation when LLMs are leveraged by prompting, and found that few-shot learning achieves the highest performance (Chapter 6).

The experimental results of the studies across publicly available datasets for defect prediction and code review automation show that

(1) PyExplainer produces (1) synthetic neighbors that are 41%-45% more similar to an instance to be explained; (2) 18%-38% more accurate local models;

and (3) explanations that are 69%-98% more unique and 17%-54% more consistent with the actual characteristics of defect-introducing commits in the future than LIME (a state-of-the-art model-agnostic technique) [22].

(2) JITLine is more accurate, faster and finer-grain than the existing deep learning-based approaches (i.e., CC2Vec [12]).

(3) DeepLineDP is 14%-24% more accurate than other file-level defect prediction approaches, and is 50%-250% more cost-effective than other line-level defect prediction approaches.

(4) The code-difference information could help D-ACT to correctly transform at least 62% more changed methods than the existing approaches.

(5) GPT-3.5 prompted by few-shot learning achieves 46.38% - 659.09% higher EM than GPT-3.5 with zero-shot learning.

The results of our study show that such contextual information being considered from various sources can help AI-powered code review assistants in generating the explanation of the prediction, generated by defect prediction models, specific to a file/patch; and substantially improve the performance of AI-based approaches for code review.

7.1 Future Work

Finally, we outline possible future research directions to leverage contextual information to explain the prediction of software defect prediction models and improve the performance of AI-based approaches for code review.

7.1.1 Exploring Other Contextual Information to Explain the Prediction of Software Defect Prediction Models

In Chapter 2 and Chapter 3, we propose the approaches that incorporate the patches similar to the patch that JIT defect prediction models predict as defective, to explain the prediction of the patch. Such approaches only rely on source code and code metrics of such patches to generate explanation. However, a patch consists of a number of elements that we do not leverage in the proposed approaches. Thus, future work can explore other information related to the predicted patch such as commit messages or results from static analysis tools [20, 21]. Using deep learning models to automatically generate the representation of such patches to generate explanation can also be explored.

In Chapter 4, we incorporate the code tokens from surrounding lines in a source code file that defect prediction models predict as defective to better localize defective lines in the file. However, source code can be represented in other forms (e.g., data flow [221] or abstract syntax trees [222]). Thus, future work can explore how to integrate other code representation to localize defective lines. The source code from other files related to the predicted file can also be utilized as contextual information.

7.1.2 Exploring Other Contextual Information for AI-based approaches for code review.

In Chapter 5, we propose the approach that incorporates the previous version of a patch to better recommend potential code improvement of the current version of the patch. In particular, the difference between the previous version of a patch and the current version of the patch is utilized to learn code improvement. Thus, future work can explore alternative or finer-grained approaches to integrate the code difference into the current version of a patch (e.g., using the difference of abstract syntax tree [223, 224]). In addition, other contextual information such as co-changed code in other files of a patch [225] or reviewers' comments of the previous version of a patch can be explored.

In Chapter 6, we incorporate code from other patches that are similar to the current version of the patch into the LLMs to better recommend potential code improvement of the current version of the patch. The study in these chapters only investigates how to utilize contextual information in the few-shot learning technique [198, 206, 207]. In addition, the similar patches are selected by using the BM25 technique [217]. Thus, future work can explore other approaches to select relevant code from other patches (e.g., using vector databases [226]) for the current version of a patch, or prompt designs that are specific to the code review automation task.

References

[1] Amiangshu Bosu, Michaela Greiler, and Christian Bird. “Characteristics of Useful Code Reviews: An Empirical Study at Microsoft”. In: *Proceedings of MSR*. 2015, pp. 146–156.

[2] Caitlin Sadowski et al. “Modern Code Review: A Case Study at Google”. In: *Proceedings of ICSE (Companion)*. 2018, pp. 181–190.

[3] Eman Abdullah AlOmar et al. “Code Review Practices for Refactoring Changes: An Empirical Study on OpenStack”. In: *arXiv preprint arXiv:2203.14404* (2022), pp. 689–701.

[4] Patanamon Thongtanunam et al. “Investigating Code Review Practices in Defective Files: An Empirical Study of the Qt System”. In: *MSR*. IEEE. 2015, pp. 168–179.

[5] Peter C Rigby and Christian Bird. “Convergent Contemporary Software Peer Review Practices”. In: *Proceedings of FSE*. 2013, pp. 202–212.

[6] Amiangshu Bosu and Jeffrey C Carver. “Impact of peer code review on peer impression formation: A survey”. In: *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE. 2013, pp. 133–142.

[7] Sunghun Kim, E James Whitehead, and Yi Zhang. “Classifying software changes: clean or buggy?” In: *Transactions on Software Engineering (TSE)* 34.2 (2008), pp. 181–196.

[8] Shivkumar Shivaji et al. “Reducing Features to Improve Code Change-Based Bug Prediction”. In: *Transactions on Software Engineering (TSE)* 39.4 (2013), pp. 552–569.

[9] Takafumi Fukushima et al. “An empirical study of just-in-time defect prediction using cross-project models”. In: *Proceedings of MSR*. 2014, pp. 172–181.

[10] Gopi Krishnan Rajbahadur et al. “The impact of using regression models to build defect classifiers”. In: *Proceedings of MSR*. Buenos Aires, Argentina, 2017, pp. 135–145.

[11] Thong Hoang et al. “DeepJIT: an end-to-end deep learning framework for just-in-time defect prediction”. In: *Proceedings of MSR*. 2019, pp. 34–45.

[12] Thong Hoang et al. “CC2Vec: Distributed representations of code changes”. In: *Proceedings of ICSE*. 2020, pp. 518–529.

[13] Michele Tufano et al. “On learning meaningful code changes via neural machine translation”. In: *Proceedings of ICSE*. 2019, pp. 25–36.

[14] Patanamon Thongtanunam, Chanathip Pornprasit, and Chakkrit Tantithamthavorn. “AutoTransform: Automated Code Transformation to Support Modern Code Review Process”. In: *Proceedings of ICSE*. 2022, pp. 237–248.

[15] Rosalia Tufan et al. “Towards automating code review activities”. In: *Proceedings of ICSE*. 2021, pp. 1479–1482.

[16] Qi Guo et al. “Exploring the Potential of ChatGPT in Automated Code Refinement: An Empirical Study”. In: *arXiv preprint arXiv:2309.08221* (2023).

[17] Zhiyu Li et al. “Automating code review activities by large-scale pre-training”. In: *Proceedings of ESEC/FSE*. 2022, pp. 1035–1047.

[18] Luca Pascarella et al. “Information needs in contemporary code review”. In: 2018, pp. 1–27.

[19] Alberto Bacchelli and Christian Bird. “Expectations, Outcomes, and Challenges of Modern Code Review”. In: *Proceedings of ICSE*. 2013, pp. 712–721.

[20] Carmine Vassallo et al. “Context is king: The developer perspective on the usage of static analysis tools”. In: *Proceedings of SANER*. 2018, pp. 38–49.

[21] Sebastiano Panichella et al. “Would static analysis tools help developers with code reviews?” In: *Proceedings of SANER*. 2015, pp. 161–170.

[22] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. “Why should I trust you?: Explaining the Predictions of Any Classifier”. In: *Proceedings of KDD*. 2016, pp. 1135–1144.
[23] Meng Yan et al. “Just-in-time defect identification and localization: A two-phase framework”. In: *IEEE Transactions on Software Engineering (TSE)* 48.1 (2020), pp. 82–101.
[24] Rosalia Tufano et al. “Using Pre-Trained Models to Boost Code Review Automation”. In: *Proceedings of ICSE*. 2022, pp. 2291–2302.
[25] Yasutaka Kamei et al. “A large-scale empirical study of just-in-time quality assurance”. In: *IEEE Transactions on Software Engineering (TSE)* 39.6 (2012), pp. 757–773.
[26] Yue Wang et al. “CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation”. In: *Proceedings of EMNLP*. 2021, pp. 8696–8708.
[27] Sunghun Kim et al. “Predicting faults from cached history”. In: *Proceedings of ICSE*. May 2007, pp. 489–498.
[28] Chanathip Pornprasit and Chakkrit Tantithamthavorn. “JITLine: A Simpler, Better, Faster, Finer-grained Just-In-Time Defect Prediction”. In: *Proceedings of MSR*. 2021, pp. 369–379.
[29] Dayi Lin, Chakkrit Tantithamthavorn, and Ahmed E Hassan. “The impact of data merging on the interpretation of cross-project just-in-time defect models”. In: *IEEE Transactions on Software Engineering (TSE)* 48.8 (2021), pp. 2969–2986.
[30] Shane McIntosh and Yasutaka Kamei. “Are Fix-Inducing Changes a Moving Target? A Longitudinal Case Study of Just-In-Time Defect Prediction”. In: *IEEE Transactions on Software Engineering (TSE)* 44.5 (2018), pp. 412–428.
[31] Hoa Khanh Dam, Truyen Tran, and Aditya Ghose. “Explainable software analytics”. In: *Proceedings of ICSE-NEIR*. 2018, pp. 53–56.
[32] Jirayus Jiarpakdee et al. “An Empirical Study of Model-Agnostic Techniques for Defect Prediction Models”. In: *IEEE Transactions on Software Engineering (TSE)* (2020), To Appear.
[33] Jirayus Jiarpakdee, Chakkrit Kla Tantithamthavorn, and John Grundy. “Practitioners’ perceptions of the goals and visual explanations of defect prediction models”. In: *Proceedings of MSR*. 2021, pp. 432–443.
[34] Chakkrit Tantithamthavorn, Jirayus Jiarpakdee, and John Grundy. “Explainable AI for Software Engineering”. In: *arXiv preprint arXiv:2012.01614* (2020).
[35] Chakkrit Tantithamthavorn, Jirayus Jiarpakdee, and John Grundy. “Actionable analytics: Stop telling me what it is; please tell me what to do”. In: *IEEE Software* 38.4 (2021), pp. 115–120.
[36] Yunzhe Jia et al. “Improving the quality of explanations with local embedding perturbations”. In: *Proceedings of SIGKDD*. 2019, pp. 875–884.
[37] Hongyu Zhang, Adam Nelson, and Tim Menzies. “On the value of learning from defect dense components for software defect prediction”. In: *Proceedings of PROMISE*. 2010, pp. 1–9.
[38] Chris Lewis et al. “Does Bug Prediction Support Human Developers? Findings from a Google Case Study”. In: *Proceedings of ICSE*. 2013, pp. 372–381.
[39] Chaiyakarn Khanan et al. “JITBot: An Explainable Just-In-Time Defect Prediction Bot”. In: *Proceedings of ASE*. 2020, pp. 1336–1339.
[40] Patanamon Thongtanunam et al. “Revisiting Code Ownership and its Relationship with Software Quality in the Scope of Modern Code Review”. In: *Proceedings of ICSE*. 2016, pp. 1039–1050.

[41] Patanamon Thongtanunam et al. “Review participation in modern code review: An empirical study of the android, Qt, and OpenStack projects”. In: *Empirical Software Engineering (EMSE)* 22 (2017), pp. 768–817.
[42] Shade Ruangwan et al. “The Impact of Human Factors on the Participation Decision of Reviewers in Modern Code Review”. In: *Empirical Software Engineering (EMSE)* 24 (2018), In press.
[43] Patanamon Thongtanunam, Weiyi Shang, and Ahmed E Hassan. “Will this clone be short-lived? Towards a better understanding of the characteristics of short-lived clones”. In: *Empirical Software Engineering (EMSE)* 24 (2019), pp. 937–972.
[44] Dong Wang et al. “Understanding shared links and their intentions to meet information needs in modern code review: A case study of the OpenStack and Qt projects”. In: *Empirical Software Engineering (EMSE)* 26 (2021), pp. 1–32.
[45] Supatsara Wattanakriengkrai et al. “Predicting defective lines using a model-agnostic technique”. In: *IEEE Transactions on Software Engineering (TSE)* 48.5 (2020), pp. 1480–1496.
[46] Thibault Laugel et al. “Defining locality for surrogates in post-hoc interpretablity”. In: *arXiv preprint arXiv:1806.07498* (2018).
[47] Rahul Krishnan, Dawen Liang, and Matthew Hoffman. “On the challenges of learning with inference networks on sparse, high-dimensional data”. In: *Proceedings of AISTATS*. PMLR. 2018, pp. 143–151.
[48] Mandavilli Srinivas and Lalit M Patnaik. “Adaptive probabilities of crossover and mutation in genetic algorithms”. In: *IEEE Transactions on Systems, Man, and Cybernetics (TSMC)* 24.4 (1994), pp. 656–667.
[49] Jerome H Friedman, Bogdan E Popescu, et al. “Predictive learning via rule ensembles”. In: *Annals of Applied Statistics* (2008), pp. 916–954.
[50] Dilini Rajapaksha et al. “SQAPlanner: Generating data-informed software quality improvement plans”. In: *IEEE Transactions on Software Engineering (TSE)* 48.8 (2021), pp. 2814–2835.
[51] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. “Anchors: High-precision model-agnostic explanations”. In: *Proceedings of AAAI*. 2018.
[52] Riccardo Guidotti et al. “Local rule-based explanations of black box decision systems”. In: *arXiv preprint arXiv:1805.10820* (2018).
[53] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. “When do changes induce fixes?” In: *Proceedings of MSR*. 2005, pp. 1–5.
[54] Suraj Yathish et al. “Mining Software Defects: Should We Consider Affected Releases?” In: *Proceedings of ICSE*. 2019, pp. 654–665.
[55] Daniel Alencar Da Costa et al. “A framework for evaluating the results of the szz approach for identifying bug-introducing changes”. In: *IEEE Transactions on Software Engineering (TSE)* 43.7 (2017), pp. 641–657.
[56] Chakkrit Tantithamthavorn et al. “An empirical comparison of model validation techniques for defect prediction models”. In: *IEEE Transactions on Software Engineering (TSE)* 43.1 (2016), pp. 1–18.
[57] Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, and Christoph Treude. “AutoSpearman: Automatically Mitigating Correlated Software Metrics for Interpreting Defect Models”. In: *Proceedings of ICSME*. 2018, pp. 92–103.

[58] Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, and Ahmed E. Hassan. “The Impact of Correlated Metrics on the Interpretation of Defect Models”. In: *IEEE Transactions on Software Engineering (TSE)* 47.2 (2021), pp. 320–331.
[59] Chakkrit Tantithamthavorn and Ahmed E. Hassan. “An Experience Report on Defect Modelling in Practice: Pitfalls and Challenges”. In: *Proceedings of ICSE-SEIP*. 2018, pp. 286–295.
[60] Helena Chmura Kraemer et al. “Measures of clinical significance”. In: *Journal of the American Academy of Child & Adolescent Psychiatry (JAACAP)* 42.12 (2003), pp. 1524–1529.
[61] John Fox. *Applied regression analysis and generalized linear models*. Sage publications, 2015.
[62] Amritanshu Agrawal and Tim Menzies. “Is Better Data Better Than Better Data Miners?: On the Benefits of Tuning SMOTE for Defect Prediction”. In: *Proceedings of ICSE*. 2018, pp. 1050–1061.
[63] Chakkrit Tantithamthavorn, Ahmed E Hassan, and Kenichi Matsumoto. “The impact of class rebalancing techniques on the performance and interpretation of defect prediction models”. In: *IEEE Transactions on Software Engineering (TSE)* 46.11 (2018), pp. 1200–1219.
[64] Nitesh V Chawla et al. “SMOTE: synthetic minority over-sampling technique”. In: *Journal of artificial intelligence research (JAIR)* 16 (2002), pp. 321–357.
[65] Guillaume LemaÃŽtre, Fernando Nogueira, and Christos K Aridas. “Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning”. In: *Journal of Machine Learning Research (JMLR)* 18.17 (2017), pp. 1–5.
[66] Chao Ni et al. “Revisiting supervised and unsupervised methods for effort-aware cross-project defect prediction”. In: *IEEE Transactions on Software Engineering (TSE)* 48.3 (2020), pp. 786–802.
[67] Kewen Peng and Tim Menzies. “Defect reduction planning (using timelime)”. In: *IEEE Transactions on Software Engineering (TSE)* 48.7 (2021), pp. 2510–2525.
[68] Jeanine Romano et al. “Appropriate Statistics for Ordinal Level Data: Should we really be using T-test and Cohen’s d for Evaluating group differences on the NSSE and other surveys”. In: *Proceedings of FAIR*. 2006, pp. 1–33.
[69] Ye Yang et al. “Actionable analytics for software engineering”. In: *IEEE Software* 35.1 (2017), pp. 51–53.
[70] Rahul Krishna and Tim Menzies. “Learning actionable analytics from multiple software projects”. In: *Empirical Software Engineering (EMSE)* 25.5 (2020), pp. 3468–3500.
[71] Di Chen et al. “Applications of Psychological Science for Actionable Analytics”. In: *Proceedings of ESEC/FSE*. 2018, pp. 456–467.
[72] Chakkrit Tantithamthavorn et al. “Automated Parameter Optimization of Classification Techniques for Defect Prediction Models”. In: *Proceedings of ICSE*. 2016, pp. 321–332.
[73] Chakkrit Tantithamthavorn et al. “The impact of automated parameter optimization on defect prediction models”. In: *IEEE Transactions on Software Engineering (TSE)* 45.7 (2018), pp. 683–711.
[74] Yasutaka Kamei et al. “Revisiting common bug prediction findings using effort-aware models”. In: *Proceedings of ICSME*. 2010, pp. 1–10.
[75] Vincent J Hellendoorn and Premkumar Devanbu. “Are deep neural networks the best choice for modeling source code?” In: *Proceedings of FSE*. 2017, pp. 763–773.
[76] Wei Fu and Tim Menzies. “Easy over hard: A case study on deep learning”. In: *Proceedings of FSE*. 2017, pp. 49–60.

[77] Zhongxin Liu et al. “Neural-machine-translation-based commit message generation: how far are we?” In: *Proceedings of ASE*. 2018, pp. 373–384.
[78] Tim Menzies et al. “500+ Times Faster than Deep Learning:(A Case Study Exploring Faster Methods for Text Mining StackOverflow)”. In: *Proceedings of MSR*. 2018, pp. 554–563.
[79] Thilo Mende and Rainer Koschke. “Effort-aware defect prediction models”. In: *Proceedings of CSMR*. 2010, pp. 107–116.
[80] Qiao Huang, Xin Xia, and David Lo. “Supervised vs unsupervised models: A holistic look at effort-aware just-in-time defect prediction”. In: *Proceedings of ICSME*. 2017, pp. 159–170.
[81] Siyuan Jiang, Ameer Armaly, and Collin McMillan. “Automatically generating commit messages from diffs using neural machine translation”. In: *Proceedings of ASE*. 2017, pp. 135–146.
[82] Luca Pascarella, Fabio Palomba, and Alberto Bacchelli. “Fine-grained just-in-time defect prediction”. In: *Journal of Systems and Software* 150 (2019), pp. 22–36.
[83] Musfiqur Rahman, Dharani Palani, and Peter C Rigby. “Natural software revisited”. In: *Proceedings of ICSE*. 2019, pp. 37–48.
[84] Wei Fu, Tim Menzies, and Xipeng Shen. “Tuning for software analytics: Is it really necessary?” In: *Information and Software Technology* 76 (2016), pp. 135–146.
[85] Rainer Storn and Kenneth Price. “Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces”. In: *Journal of global optimization* 11 (1997), pp. 341–359.
[86] Pauli Virtanen et al. “SciPy 1.0: fundamental algorithms for scientific computing in Python”. In: *Nature methods* 17.3 (2020), pp. 261–272.
[87] Amritanshu Agrawal et al. “How to “dodge” complex software analytics”. In: *IEEE Transactions on Software Engineering (TSE)* 47.10 (2019), pp. 2182–2194.
[88] Gema Rodríguez-Pérez, Gregorio Robles, and Jesús M González-Barahona. “Reproducibility and credibility in empirical software engineering: A case study based on a systematic literature review of the use of the szz algorithm”. In: *Information and Software Technology (IST)* 99 (2018), pp. 164–176.
[89] Matthieu Jimenez et al. “The importance of accounting for real-world labelling when predicting software vulnerabilities”. In: *Proceedings of FSE*. 2019, pp. 695–705.
[90] Yibiao Yang et al. “Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models”. In: *Proceedings of FSE*. 2016, pp. 157–168.
[91] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. “PyDriller: Python Framework for Mining Software Repositories”. In: *Proceedings of ESEC/FSE*. Lake Buena Vista, FL, USA, 2018, pp. 908–911.
[92] Jirayus Jiarpakdee et al. “An empirical study of model-agnostic techniques for defect prediction models”. In: *IEEE Transactions on Software Engineering* 48.1 (2020), pp. 166–185.
[93] Ming Tan et al. “Online defect prediction for imbalanced data”. In: *Proceedings of ICSE*. 2015, pp. 99–108.
[94] George G Cabral et al. “Class imbalance evolution and verification latency in just-in-time software defect prediction”. In: *Proceedings of ICSE*. 2019, pp. 666–676.
[95] Cynthia Liem and Annibale Panichella. “Run, Forest, Run? On Randomization and Reproducibility in Predictive Software Engineering”. In: *arXiv preprint arXiv:2012.08387* (2020).

[96] Carmine Vassallo et al. “Continuous Code Quality: Are We (Really) Doing That?” In: *Proceedings of ASE*. 2018, pp. 790–795.
[97] Hoa Khanh Dam et al. “Automatic feature learning for predicting vulnerable software components”. In: *IEEE Transactions on Software Engineering (TSE)* 47.1 (2018), pp. 67–85.
[98] Song Wang, Taiyue Liu, and Lin Tan. “Automatically Learning Semantic Features for Defect Prediction”. In: *Proceedings of ICSE*. 2016, pp. 297–308.
[99] Jundong Li et al. “Feature selection: A data perspective”. In: *ACM computing surveys (CSUR)* 50.6 (2017), pp. 1–45.
[100] Y. Kamei et al. “The Effects of Over and Under Sampling on Fault-prone Module Detection”. In: *Proceedings of ESEM*. 2007, pp. 196–204.
[101] Hideaki Hata, Osamu Mizuno, and Tohru Kikuno. “Bug prediction based on fine-grained module histories”. In: *Proceedings of ICSE*. 2012, pp. 200–210.
[102] Song Wang et al. “Deep semantic feature learning for software defect prediction”. In: *IEEE Transactions on Software Engineering* 46.12 (2018), pp. 1267–1293.
[103] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. “Neural machine translation by jointly learning to align and translate”. In: *arXiv preprint arXiv:1409.0473* (2014).
[104] Zichao Yang et al. “Hierarchical attention networks for document classification”. In: *Proceedings of NAACL*. 2016, pp. 1480–1489.
[105] Ashish Vaswani et al. “Attention is All You Need”. In: *Proceedings of NIPS*. 2017, pp. 5999–6009.
[106] Barbara A Kitchenham and Shari L Pfleeger. “Personal opinion surveys”. In: *Guide to Advanced Empirical Software Engineering*. 2008, pp. 63–92.
[107] Haitao Mi, Zhiguo Wang, and Abe Ittycheriah. “Vocabulary manipulation for neural machine translation”. In: *arXiv preprint arXiv:1605.03209* (2016).
[108] Martin White et al. “Toward deep learning software repositories”. In: *Proceedings of MSR*. 2015, pp. 334–345.
[109] Guillaume Bouchard. “Efficient bounds for the softmax function, applications to inference in hybrid models”. In: *Proceedings of NIPS*. 2007.
[110] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. “Mining Metrics to Predict Component Failures”. In: *Proceedings of ICSE*. 2006, pp. 452–461.
[111] Foyzur Rahman et al. “Comparing static bug finders and statistical prediction”. In: *Proceedings of ICSE*. 2014, pp. 424–434.
[112] Baishakhi Ray et al. “On the Naturalness of Buggy Code”. In: *Proceedings of ICSE*. 2016, pp. 428–439.
[113] Diederik P Kingma and Jimmy Ba. “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980* (2014).
[114] Cody Watson et al. “A Systematic Literature Review on the Use of Deep Learning in Software Engineering Research”. In: *arXiv preprint arXiv:2009.06520* (2020).
[115] Chris Parnin and Alessandro Orso. “Are automated debugging techniques actually helping programmers?” In: *Proceedings of ISSTA*. 2011, pp. 199–209.
[116] Hideaki Hata, Osamu Mizuno, and Tohru Kikuno. “Fault-prone module detection using large-scale text features based on spam filtering”. In: *Empirical Software Engineering (EMSE)* 15 (2010), pp. 147–165.
[117] Jian Li et al. “Software defect prediction via convolutional neural network”. In: *Proceedings of QRS*. 2017, pp. 318–328.
[118] Yi Li et al. “Improving bug detection via context-based code representation learning and attention-based neural networks”. In: *Proceedings of PACMPL* (2019), pp. 1–30.

[119] Guanjun Lin et al. “Cross-project transfer representation learning for vulnerable function discovery”. In: *IEEE Transactions on Industrial Informatics* 14.7 (2018), pp. 3289–3297.
[120] Deqing Zou et al. “μVulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection”. In: *IEEE Transactions on Dependable and Secure Computing (TDSC)* 18.5 (2019), pp. 2224–2236.
[121] Zhen Li et al. “Vuldeepecker: A deep learning-based system for vulnerability detection”. In: *arXiv preprint arXiv:1801.01681* (2018).
[122] Song Wang et al. “Bugram: Bug Detection with N-gram Language Models”. In: *Proceedings of ASE*. 2016, pp. 708–719.
[123] Andrew Habib and Michael Pradel. “How many of all bugs do we find? a study of static bug detectors”. In: *Proceedings of ASE*. 2018, pp. 317–328.
[124] E. Aftandilian et al. “Building Useful Program Analysis Tools Using an Extensible Java Compiler”. In: *Proceedings of SCAM*. 2012, pp. 14–23.
[125] Thomas Zimmermann et al. “Cross-project defect prediction: a large scale experiment on data vs. domain vs. process”. In: *Proceedings of FSE*. 2009, pp. 91–100.
[126] Tim Menzies et al. “Local vs. global models for effort estimation and defect prediction”. In: *Proceedings of ASE* (2011), pp. 343–351.
[127] Feng Zhang et al. “Cross-project defect prediction using a connectivity-based unsupervised classifier”. In: *Proceedings of ICSE*. 2016, pp. 309–320.
[128] Xin Xia et al. “Hydra: Massively compositional model for cross-project defect prediction”. In: *IEEE Transactions on software Engineering (TSE)* 42.10 (2016), pp. 977–998.
[129] Burak Turhan et al. “On the relative value of cross-company and within-company data for defect prediction”. In: *Empirical Software Engineering (ESEM)* 14 (2009), pp. 540–578.
[130] Zhiyuan Wan et al. “Perceptions, expectations, and challenges in defect prediction”. In: *IEEE Transactions on Software Engineering (TSE)* 46.11 (2018), pp. 1241–1266.
[131] Erik Arisholm, Lionel C Briand, and Eivind B Johannessen. “A systematic and comprehensive investigation of methods to build and evaluate fault prediction models”. In: *Journal of Systems and Software (JSS)* 83.1 (2010), pp. 2–17.
[132] Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, and Christoph Treude. “The impact of automated feature selection techniques on the interpretation of defect models”. In: *Empirical Software Engineering (ESEM)* 25 (2020), pp. 3590–3638.
[133] Tim Menzies, Jeremy Greenwald, and Art Frank. “Data mining static code attributes to learn defect predictors”. In: *IEEE transactions on software engineering (TSE)* 33.1 (2007), pp. 2–13.
[134] Fumio Akiyama. “An Example of Software System Debugging”. In: *IFIP Congress*. Vol. 71. 1971, pp. 353–359.
[135] Victor R Basili, Lionel C. Briand, and Walcélio L Melo. “A validation of object-oriented design metrics as quality indicators”. In: *IEEE Transactions on software engineering (TSE)* 22.10 (1996), pp. 751–761.
[136] Fabio Palomba et al. “Mining version histories for detecting code smells”. In: *IEEE Transactions on Software Engineering (TSE)* 41.5 (2014), pp. 462–489.
[137] Davide Spadini et al. “On the relation of test smells to software code quality”. In: *Proceedings of ICSME*. 2018, pp. 1–12.
[138] Nachiappan Nagappan and Thomas Ball. “Use of Relative Code Churn Measures to Predict System Defect Density”. In: *Proceedings of ICSE*. 2005, pp. 284–292.
[139] Christian Bird et al. “Putting it All Together: Using Socio-technical Networks to Predict Failures”. In: *Proceedings of ISSRE* (2009), pp. 109–119.

[140] Kim Herzig. “Using pre-release test failures to build early post-release defect prediction models”. In: *Proceedings of ISSRE*. 2014, pp. 300–311.
[141] Foyzur Rahman and Premkumar Devanbu. “Ownership, experience and defects: a fine-grained study of authorship”. In: *Proceedings of ICSE*. 2011, pp. 491–500.
[142] Ahmed E Hassan and Richard C Holt. “The Top Ten List: Dynamic Fault Prediction”. In: *Proceedings of ICSM*. 2005, pp. 263–272.
[143] Ahmed E. Hassan. “Predicting Faults using the Complexity of Code Changes”. In: *Prooceedings of ICSE*. 2009, pp. 78–88.
[144] Dario Di Nucci et al. “A developer centered bug prediction model”. In: *IEEE Transactions on Software Engineering (TSE)* 44.1 (2017), pp. 5–24.
[145] Amirabbas Majd et al. “SLDeep: Statement-level software defect prediction using deep-learning model on static code features”. In: *Expert Systems with Applications* 147 (2020), p. 113156.
[146] Brittany Johnson et al. “Why don’t software developers use static analysis tools to find bugs?” In: *Proceedings of ICSE*. 2013, pp. 672–681.
[147] Chanathip Pornprasit et al. “PyExplainer: Explaining the Predictions of Just-In-Time Defect Models”. In: *Proceedings of ASE*. 2021, pp. 407–418.
[148] Jakub Lipcak and Bruno Rossi. “A large-scale study on source code reviewer recommendation”. In: *Proceedings of SEAA*. 2018, pp. 378–387.
[149] Xiaofeng Han et al. “Code smells detection via modern code review: A study of the openstack and qt communities”. In: *Empirical Software Engineering (ESEM)* 27.6 (2022), p. 127.
[150] Laura MacLeod et al. “Code Reviewing in the Trenches: Challenges and Best Practices”. In: *IEEE Software* (2017), pp. 34–42.
[151] Rico Sennrich, Barry Haddow, and Alexandra Birch. “Neural Machine Translation of Rare Words with Subword Units”. In: *Proceedings of ACL*. 2016, pp. 1715–1725.
[152] Moritz Beller et al. “Modern code reviews in open-source projects: Which problems do they fix?” In: *Proceedings of MSR*. 2014, pp. 202–211.
[153] Guowu Xie, Jianbo Chen, and Iulian Neamtiu. “Towards a better understanding of software evolution: An empirical study on open source software”. In: *Proceedings of ICSM*. 2009, pp. 51–60.
[154] Yue Liu et al. “Explainable AI for Android Malware Detection: Towards Understanding Why the Models Perform So Well?” In: *Proceedings of ISSRE*. 2022, pp. 169–180.
[155] Zhangyin Feng et al. “Codebert: A pre-trained model for programming and natural languages”. In: *arXiv preprint arXiv:2002.08155* (2020).
[156] Daya Guo et al. “Graphcodebert: Pre-training code representations with data flow”. In: *arXiv preprint arXiv:2009.08366* (2020).
[157] Wasi Ahmad et al. “Unified Pre-training for Program Understanding and Generation”. In: *Proceedings of NAACL*. 2021, pp. 2655–2668.
[158] *A Replication Package*. https://github.com/awsm-research/D-ACT-Replication-Package.
[159] Yuanrui Fan et al. “Early Prediction of Merged Code Changes to Prioritize Reviewing Tasks”. In: *Empirical Software Engineering (EMSE)* 23.6 (2018), pp. 3346–3393.
[160] Tobias Baum, Kurt Schneider, and Alberto Bacchelli. “Associating Working Memory Capacity and Code Change Ordering with Code Review Performance”. In: *Empirical Software Engineering (EMSE)* 24.4 (2019), pp. 1762–1798.
[161] Michael Fu and Chakkrit Tantithamthavorn. “LineVul: A Transformer-based Line-Level Vulnerability Prediction”. In: *Proceedings of MSR*. 2022, pp. 608–620.

[162] Chanathip Pornprasit and Chakkrit Kla Tantithamthavorn. “Deeplinedp: Towards a deep learning approach for line-level defect prediction”. In: *IEEE Transactions on Software Engineering* 49.1 (2022), pp. 84–98.
[163] Vipin Balachandran. “Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation”. In: *Proceedings of ICSE*. IEEE. 2013, pp. 931–940.
[164] Christoph Hannebauer et al. “Automatically recommending code reviewers based on their expertise: An empirical comparison”. In: *Proceedings of ASE*. 2016, pp. 99–110.
[165] Patanamon Thongtanunam et al. “Who Should Review My Code? A File Location-based Code-reviewer Recommendation Approach for Modern Code Review”. In: *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 2015, pp. 141–150.
[166] Motahareh Bahrami Zanjani, Huzefa Kagdi, and Christian Bird. “Automatically recommending peer reviewers in modern code review”. In: 42.6 (2015), pp. 530–543.
[167] Wisam Haitham Abbood Al-Zubaidi et al. “Workload-aware reviewer recommendation using a multi-objective search-based approach”. In: *Proceedings of PROMISE*. 2020, pp. 21–30.
[168] Yang Hong et al. “CommentFinder: a simpler, faster, more accurate code review comments recommendation”. In: *Proceedings of ESEC/FSE*. 2022, pp. 507–519.
[169] Yang Hong, Chakkrit Tantithamthavorn, and Patanamon Thongtanunam. “Where Should I Look at? Recommending Lines that Reviewers Should Pay Attention To”. In: *Proceedings of SANER*. 2022, pp. 1034–1045.
[170] Colin Raffel et al. “Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer”. In: *Journal of Machine Learning Research (JMLR)* 21.140 (2020), pp. 1–67.
[171] Fuzhen Zhuang et al. “A comprehensive survey on transfer learning”. In: *Proceedings of the IEEE* 109.1 (2020), pp. 43–76.
[172] Yue Liu et al. “AutoUpdate: Automatically Recommend Code Updates for Android Apps”. In: *arXiv preprint arXiv:2209.07048* (2022).
[173] Veit Frick et al. “Generating accurate and compact edit scripts using tree differencing”. In: *Proceedings of ICSME*. 2018, pp. 264–274.
[174] Zhongxin Liu et al. “Just-In-Time Obsolete Comment Detection and Update”. In: *IEEE Transactions of Software Engineering (TSE)* 49.1 (2021), pp. 1–23.
[175] Rafael-Michael Karampatsis et al. “Big code!= big vocabulary: Open-vocabulary models for source code”. In: *Proceedings of ICSE*. 2020, pp. 1073–1085.
[176] Michael Fu et al. “VulRepair: a T5-based automated software vulnerability repair”. In: *Proceedings of ESEC/FSE*. 2022, pp. 935–947.
[177] Hamel Husain et al. “Codesearchnet challenge: Evaluating the state of semantic code search”. In: *arXiv preprint arXiv:1909.09436* (2019).
[178] Jacob Devlin et al. “Bert: Pre-training of deep bidirectional transformers for language understanding”. In: *arXiv preprint arXiv:1810.04805* (2018).
[179] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. “Layer normalization”. In: *arXiv preprint arXiv:1607.06450* (2016).
[180] Kaiming He et al. “Deep residual learning for image recognition”. In: *Proceedings of CVPR*. 2016, pp. 770–778.
[181] Thomas Wolf et al. “Huggingface’s transformers: State-of-the-art natural language processing”. In: *arXiv preprint arXiv:1910.03771* (2019).

[182] Ilya Loshchilov and Frank Hutter. “Decoupled weight decay regularization”. In: *arXiv preprint arXiv:1711.05101* (2017).
[183] Kishore Papineni et al. “BLEU: A Method for Automatic Evaluation of Machine Translation”. In: *Proceedings of ACL*. 2002, pp. 311–318.
[184] Yangruibo Ding et al. “Patching as Translation: The Data and the Metaphor”. In: *Proceedings of ASE*. 2020, pp. 275–286.
[185] Shuo Ren et al. “Codebleu: a method for automatic evaluation of code synthesis”. In: *arXiv preprint arXiv:2009.10297* (2020).
[186] Mike Lewis et al. “BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension”. In: *Proceedings of ACL*. 2020, pp. 7871–7880.
[187] Alec Radford et al. “Language models are unsupervised multitask learners”. In: *OpenAI blog* (2019), p. 9.
[188] Rosalia Tufano et al. “Towards automating code review activities”. In: *Proceedings of ICSE*. 2021, pp. 163–174.
[189] Chanathip Pornprasit et al. “D-ACT: Towards Diff-Aware Code Transformation for Code Review Under a Time-Wise Evaluation”. In: *Proceedings of SANER*. 2023, pp. 296–307.
[190] J. Lu et al. “LLaMA-Reviewer: Advancing Code Review Automation with Large Language Models through Parameter-Efficient Fine-Tuning”. In: *Proceedings of ISSRE*. 2023, pp. 647–658.
[191] Hugo Touvron et al. “Llama: Open and efficient foundation language models”. In: *arXiv preprint arXiv:2302.13971* (2023).
[192] Shuzheng Gao et al. “What Makes Good In-Context Demonstrations for Code Intelligence Tasks with LLMs?” In: *Proceedings of ASE*. 2023, pp. 761–773.
[193] Shuzheng Gao et al. “Constructing Effective In-Context Demonstration for Code Intelligence Tasks: An Empirical Study”. In: *Proceedings of ASE*. 2023.
[194] Jules White et al. “A prompt pattern catalog to enhance prompt engineering with chatgpt”. In: *arXiv preprint arXiv:2302.11382* (2023).
[195] Aakanksha Chowdhery et al. “Palm: Scaling language modeling with pathways”. In: *Journal of Machine Learning Research (JMLR)* (2023), pp. 1–113.
[196] Jason Wei et al. “Finetuned language models are zero-shot learners”. In: *arXiv preprint arXiv:2109.01652* (2021).
[197] Mark Chen et al. “Evaluating large language models trained on code”. In: *arXiv preprint arXiv:2107.03374* (2021).
[198] Tom Brown et al. “Language models are few-shot learners”. In: *Proceedings of NeurIPS* (2020), pp. 1877–1901.
[199] Sungmin Kang, Juyeon Yoon, and Shin Yoo. “Large language models are few-shot testers: Exploring llm-based general bug reproduction”. In: *Proceedings of ICSE*. 2023, pp. 2312–2323.
[200] Mingyang Geng et al. “Large Language Models are Few-Shot Summarizers: Multi-Intent Comment Generation via In-Context Learning”. In: *Proceedings of ICSE*. 2024, pp. 1–13.
[201] Yuxiang Wei et al. “Magicoder: Source code is all you need”. In: *arXiv preprint arXiv:2312.02120* (2023).
[202] *Supplementary material*. https://github.com/awsm-research/LLM-for-code-review-automatiton.
[203] Baptiste Rozière et al. “Code Llama: Open Foundation Models for Code”. In: *arXiv preprint arXiv:2308.12950* (2023).

[204] Raymond Li et al. “Starcoder: may the source be with you!” In: *arXiv preprint arXiv:2305.06161* (2023).
[205] Hugo Touvron et al. “Llama 2: Open foundation and fine-tuned chat models”. In: *arXiv preprint arXiv:2307.09288* (2023).
[206] Jiachang Liu et al. “What Makes Good In-Context Examples for GPT-3?” In: *arXiv preprint arXiv:2101.06804* (2021).
[207] Jerry Wei et al. “Larger language models do in-context learning differently”. In: *arXiv preprint arXiv:2303.03846* (2023).
[208] Seungone Kim et al. “The cot collection: Improving zero-shot and few-shot learning of language models via chain-of-thought fine-tuning”. In: *arXiv preprint arXiv:2305.14045* (2023).
[209] Jason Wei et al. “Chain-of-thought prompting elicits reasoning in large language models”. In: *NeruIIPS* (2022), pp. 24824–24837.
[210] Xuezhi Wang et al. “Self-consistency improves chain of thought reasoning in language models”. In: *arXiv preprint arXiv:2203.11171* (2022).
[211] Junjielong Xu et al. “Prompting for Automatic Log Template Extraction”. In: *arXiv preprint arXiv:2307.09950* (2023).
[212] Malinda Dilhara et al. “Unprecedented Code Change Automation: The Fusion of LLMs and Transformation by Example”. In: *arXiv preprint arXiv:2402.07138* (2024).
[213] Md Rakib Hossain Misu et al. “Towards AI-Assisted Synthesis of Verified Dafny Methods”. In: *arXiv preprint arXiv:2402.00247* (2024).
[214] Zhihan Jiang et al. “Llmparser: A llm-based log parsing framework”. In: *arXiv preprint arXiv:2310.01796* (2023).
[215] Xin Zhou et al. “Generation-based Code Review Automation: How Far Are We?” In: *arXiv preprint arXiv:2303.07221* (2023).
[216] Shih-Yang Liu et al. “DoRA: Weight-Decomposed Low-Rank Adaptation”. In: *arXiv preprint arXiv:2402.09353* (2024).
[217] Stephen Robertson, Hugo Zaragoza, et al. “The probabilistic relevance framework: BM25 and beyond”. In: *Foundations and Trends® in Information Retrieval* (2009), pp. 333–389.
[218] Zhiqiang Yuan et al. “Evaluating instruction-tuned large language models on code comprehension and generation”. In: *arXiv preprint arXiv:2308.01240* (2023).
[219] David OBrien et al. “Are Prompt Engineering and TODO Comments Friends or Foes? An Evaluation on GitHub Copilot”. In: *Proceedings of ICSE*. 2024, pp. 1–13.
[220] Soneya Binta Hossain et al. “A deep dive into large language models for automated bug localization and repair”. In: *arXiv preprint arXiv:2404.11595* (2024).
[221] Lloyd D Fosdick and Leon J Osterweil. “Data flow analysis in software reliability”. In: *ACM Computing Surveys (CSUR)* 8.3 (1976), pp. 305–330.
[222] Christopher A Welty. “Augmenting abstract syntax trees for program understanding”. In: *Proceedings of ASE*. 1997, pp. 126–133.
[223] Iulian Neamtiu, Jeffrey S Foster, and Michael Hicks. “Understanding source code evolution using abstract syntax tree matching”. In: *Proceedings of MSR*. 2005, pp. 1–5.
[224] Pouria Alikhanifard and Nikolaos Tsantalis. “A novel refactoring and semantic aware abstract syntax tree differencing tool and a benchmark for evaluating the accuracy of diff tools”. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* (2024).
[225] Luciana L Silva, Marco Tulio Valente, and Marcelo A Maia. “Co-change patterns: A large scale empirical study”. In: *Journal of Systems and Software (JSS)* 152 (2019), pp. 196–214.

[226] Jasper Xian et al. “Vector search with OpenAI embeddings: Lucene is all you need”. In: *Proceedings of WSDM*. 2024, pp. 1090–1093.