PolyPublie
Polytechnique Montréal

POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

Titre: Title:	Enhancing Software Engineering Tasks with Intelligent Assistant Tools
Auteur: Author:	Arghavan Moradidakhel
Date:	2023
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Moradidakhel, A. (2023). Enhancing Software Engineering Tasks with Intelligent Assistant Tools [Thèse de doctorat, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/57120/

Document en libre accès dans PolyPublie
Open Access document in PolyPublie

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/57120/
Directeurs de recherche: Advisors:	Michel Desmarais, David Barrera, & Foutse Khomh
Programme: Program:	Génie informatique

Ce fichier a été téléchargé à partir de PolyPublie, le dépôt institutionnel de Polytechnique Montréal
This file has been downloaded from PolyPublie, the institutional repository of Polytechnique Montréal

https://publications.polymtl.ca

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Enhancing Software Engineering Tasks with Intelligent Assistant Tools

ARGHAVAN MORADIDAKHEL
Département de génie informatique et génie logiciel

Thèse présentée en vue de l'obtention du diplôme de *Philosophiæ Doctor*
Génie informatique

Décembre 2023

© Arghavan Moradidakhel, 2023.

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Cette thèse intitulée :

Enhancing Software Engineering Tasks with Intelligent Assistant Tools

présentée par **Arghavan MORADIDAKHEL**
en vue de l'obtention du diplôme de *Philosophiæ Doctor*
a été dûment acceptée par le jury d'examen constitué de :

Christopher J. PAL, président
Michel DESMARAIS, membre et directeur de recherche
Foutse KHOMH, membre et codirecteur de recherche
David BARRERA, membre et codirecteur de recherche
Laurent CHARLIN, membre
Gabriele BAVOTA, membre externe

DEDICATION

Them that I love, they know that I love them. Thus,...
To all remarkable women in STEM—true warriors who shattered barriers and paved the way for future generations....

ACKNOWLEDGEMENTS

The completion of this study could not have been possible without the expertise and support of the following people.

I would like to thank Professor Foutse and Professor Michel for teaching me the wisdom of knowing when to search for a key to unlock a door and when to bravely accept a closed door and forge a new path. Your unwavering belief in me over these five years has been a constant source of inspiration and encouragement. Thank you for showing me the joys of research.

To my mom & dad, and my only sister, Gilda. Thank you for your unconditional love and support through all of these years. I know that it hasn't been easy to support me with all the distances between us.

To Kasra, who sometimes talks, usually listens, and always cares—your warmest support and empathetic ear helped me endure the exhaustive hours of work.

To Elaheh & Peji, true friends I found on this journey (may we all find a few...). Your friendship has turned the thunders of this journey into a whisper.

I would also like to thank my co-authors who collaborated with me on the studies in this thesis and the reviewers, the anonymous heroes, who brought new ideas and proposed improvements to our studies.

Additionally, I would like to thank my committee members, Professors Gabriele BAVOTA, Laurent CHARLIN, and Christopher J. PAL, for evaluating my thesis.

RÉSUMÉ

Au fil des décennies, diverses recherches ont été entreprises pour développer des outils d'assistance pour soutenir les équipes de développement logiciel à différentes étapes du cycle de développement logiciel, allant de l'ingénierie des exigences aux tâches liées aux tests et à la maintenance du logiciel. L'importance de ces outils découle de leur impact sur la productivité et l'éfficacité des développeurs et, par conséquent, sur la qualité du logiciel. Cependant, il est crucial de noter que la fiabilité et l'efficacité de ces outils d'assistance jouent un rôle critique dans la confiance que les développeurs accordent à ces outils ainsi que dans leur adoption. Par conséquent, il est essentiel de bien comprendre les limites de ces outils et de savoir les utiliser judicieusement.

De nos jours, des plateformes telles que GitHub hébergent une abondance de données de développement logiciel. Ces données offrent de belles opportunités pour la création de méthodes et d'outils d'assistance aux développeurs logiciels basés sur les données, notamment l'apprentissage automatique (ML) et l'apprentissage profond (DL). Ces approches peuvent révolutionner la création d'outils d'assistance intelligents en génie logiciel (SE) et l'émergence des grands modèles de langage (LLMs) a marqué un tournant important récemment.

Dans cette thèse, nous proposons des méthodologies qui utilisent l'apprentissage statistique pour améliorer la justesse et l'efficacité des outils d'assistance intelligents. Trois tâches cruciales du génie logiciel (SE) sont visées : la génération de code, la génération de cas de test, et l'identification d'experts. Nos études sont inspirées par les limitations que nous avons observées dans les outils d'assistance de pointe qui existent présentement. Nous tirons des enseignements des techniques établies en génie logiciel et des théories pertinentes, et proposons des méthodes qui intègrent les capacités des techniques d'analyse de code avec le pouvoir génératif des LLMs, pour répondre à ces limitations.

Dans notre première étude, nous évaluons la qualité du code généré par un outil d'assistance à la programmation incorporant un LLM, Github Copilot, en le comparant à des humains pour la tâche de génération de code. Nous mettons en lumière ses limitations et ses capacités, explorant comment, malgré ses limites, il peut efficacement servir d'outil de programmation en binôme. Nos résultats révèlent que bien que le code généré par des LLMs puisse contenir plus de bogues que celui élaboré par des développeurs novices humains, le coût de réparation du code défectueux généré par un LLM est plus bas selon différentes mesures. S'appuyant sur nos découvertes, nous proposons l'utilisation d'un outil de réparation de code défectueux afin d'accroître la fiabilité du code généré par un outil d'assistance à la programmation basé

sur les LLMs, réduisant ainsi le besoin de vérification et d'intervention humaine. De plus, nous construisons une taxonomie des bogues rencontrés fréquement dans du code généré par LLMs. Cette taxonomie peut constituer une ressource précieuse pour des études futures visant à améliorer les outils d'assistance et les techniques d'assurance qualité, telles que les approches de *Mutation Testing* (MT), afin de leur permettre de s'adapter aux caractéristiques des bogues qui se manifestent dans le code généré par LLMs.

Dans notre deuxième étude, nous proposons une méthode qui intègre les LLMs pour générer des cas de test efficaces. Nous commençons par évaluer l'efficacité des cas de test générés par les LLMs pour révéler des bogues, en utilisant du Mutation Testing (MT). Il s'agit de la première étude à utiliser le MT pour évaluer l'efficacité des cas de test générés par les LLMs pour la détection de bogues. Notre méthodologie comprend plusieurs étapes de post-traitement visant à réduire l'effort nécessaire pour valider la correction des cas de test et évaluer leur efficacité. Sur la base de cette évaluation, nous proposons une approche d'apprentissage basée sur l'utilisation de "prompt" contenant des "mutants survivants". Les résultats de nos analyses montrent que cette approche permet d'améliorer l'efficacité des cas de test générés par les LLMs, surpassant diverses techniques de référence et outils de pointe provenant de la litérature et de l'industrie.

Une autre tâche essentielle dans le développement et la maintenance de logiciels est l'identification d'experts possédant les compétences nécessaires pour une tâche ou un projet spécifique. Une évaluation fiable de l'expertise des développeurs est une étape cruciale pour effectuer une assignation de développeurs à des tâches qui relève de leur champ d'expertise. Une assignation adéquate contribue à l'exécution efficace de diverses activités dans des délais optimaux. Ainsi, dans les deux dernières études de cette thèse, nous proposons et évaluons des modèles de représentation et d'évaluation de l'expertise des développeurs à partir de leurs contributions à des projets open source.

Dans notre troisième étude, nous proposons une méthode permettant de représenter l'expertise des développeurs dans un espace de plongements vectoriels ("embedding space"). Une représentation numérique de l'expertise des développeurs est essentielle pour diverses techniques d'apprentissage automatique, notamment la classification des développeurs dans différentes catégories ou la sélection de candidats pour des tâches spécifiques. Nous utilisons *doc2vec* et introduisons une méthode pour représenter l'expertise des développeurs dans un espace de plongements. Par rapport aux techniques de pointe existantes, nos résultats démontrent que cette méthode offre des performances supérieures pour l'évaluation de l'expertise et du domaine de spécialisation des développeurs.

Dans notre quatrième étude, nous proposons une approche pour modéliser l'expertise des

développeurs en se basant sur la distribution des motifs syntaxiques dans leur code. Ce concept s'inspire d'une théorie linguistique connue sous le nom de loi de Zipf. Nos résultats révèlent que la distribution des motifs syntaxiques dans le code rédigé par des experts peut être distinguée de celui des novices. La distribution Zipf du code des experts présente une queue plus longue, tandis que dans le code des novices, elle présente un sommet plus large, signifiant des motifs plus répétitifs.

Les résultats présentés dans cette thèse représentent des étapes importantes vers l'amélioration de l'efficacité et de la fiabilité des outils d'assistance développés pour soutenir les équipes de développement logiciel à différentes étapes du cycle de développement du logiciel. Nous croyons que les limites que nous avons identifiées, ainsi que les méthodes proposées pour les résoudre, aideront les chercheurs et les praticiens à améliorer la fiabilité de ces outils.

Les résultats présentés dans cette thèse représentent des étapes importantes visant à améliorer la précision et l'efficacité des outils d'assistance intelligents. Nous croyons que les connaissances que nous avons identifiées aideront les chercheurs et les praticiens à automatiser différentes tâches en génie logiciel.

ABSTRACT

Over the decades, various research has been done on developing assistant tools for various stages of the software development lifecycle, ranging from requirement engineering to tasks related to software testing and maintenance. The importance of these tools stems from their impact on developers' productivity and, consequently, on software quality. However, it is crucial to note that the correctness, and effectiveness of these assistant tools play a pivotal role in shaping developers' trust and willingness to integrate them. This underscores the importance of comprehending the limitations of these tools and developing methods that can be incorporated to improve them.

Nowadays, platforms such as GitHub host an abundance of data related to various types of code development. This data presents significant potential for proposing methodologies to enhance data-driven assistant tools for software development tasks. Additionally, Machine Learning (ML) and Deep Learning (DL) have introduced new possibilities for developing diverse methods that can revolutionize the creation of intelligent assistant tools in Software Engineering (SE). A noteworthy milestone in this evolution is the emergence of Large Language Models (LLMs).

In this thesis, we propose methodologies that employ statistical learning to enhance the correctness and effectiveness of intelligent assistant tools in three pivotal tasks within SE: code generation, test case generation, and expert identification. Our studies are inspired by the limitations that we observed in state-of-the-art assistant tools. We draw insights from established SE techniques and relevant theories and propose methods that incorporate the capabilities of ML/DL techniques, alongside the generative power of LLMs, to address these limitations.

In our first study, we evaluate the quality of code generated by an AI programming assistant tool incorporating LLM, comparing it to human-generated code. We shed light on its limitations and capabilities and explore how, despite its limitations, it can effectively serve as a pair programming tool. Our findings reveal that while LLM-generated code may contain more bugs than code crafted by human novices, the cost of repairing LLM-generated buggy code is lower in terms of different metrics. Building on our findings, we propose the use of a programming repair tool to increase the reliability of code generated by an LLM-based programming assistant tool, thereby diminishing the need for human verification and intervention. Additionally, we extract a taxonomy of bugs in code generated by LLMs that can serve as a valuable resource for future studies to enhance assistant tools and techniques, such

as Mutation Testing (MT) approaches, to adapt them to the characteristics of the bugs that arise in code generated by LLMs.

In our second study, we focus on developing a method that integrates LLMs to generate effective test cases. We start by evaluating the effectiveness of the test cases generated by LLMs in revealing bugs by using Mutation Testing (MT). This is the first study to employ MT to evaluate the effectiveness of test cases generated by LLMs in revealing bugs. Our methodology involves several post-processing steps aimed at minimizing the effort required to validate test cases' correctness and assess their effectiveness. Based on this evaluation, we propose a prompt-based learning approach that uses surviving mutants to enhance the effectiveness of LLM-generated test cases. Our proposed method outperforms various baselines and state-of-the-art tools.

An essential task in software development and maintenance is identifying an expert with the necessary skills for a specific task or project. Accurate assessments of developers' expertise are crucial steps toward assigning developers to the right tasks, contributing to the efficient execution of various activities within optimal timeframes. Hence, in the next two studies of this thesis, we explore the representation and the assessment of developers' expertise based on their contributions to open-source projects.

In our third study, we shift our attention to proposing a method to represent the domain expertise of developers in an embedding space. Numerical representation of the domain expertise of developers is essential for various ML techniques, including developer classification into different domains or the selection of candidates for specific tasks. We utilize *doc2vec* and introduce a method for representing the domain expertise of developers in an embedding space. Our findings demonstrate superior performance in representing developers' expertise and assessing their technical specialization compared to state-of-the-art techniques. The results of this study have applications as a step towards diverse downstream tasks, such as developer task/issue assignment and job posting candidate selection.

In our fourth study, we propose an approach to modeling the expertise of developers based on the distribution of syntactical patterns in their code. This concept draws inspiration from a linguistic theory known as Zipf's law. Our result reveals that the distribution of syntax patterns in code written by experts can be distinguished from novices. This distribution in experts' code exhibits a longer tail, while in novices' code shows a wider vertex, signifying more repetitive patterns.

The results presented in this thesis are important steps toward enhancing the correctness and effectiveness of intelligent assistant tools. We believe that the insights we have identified will assist researchers and practitioners in automating different SE tasks.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ACRONYMS

SE	Software Engineering
ML	Machine Learning
DL	Deep Learning
LLM	Large Language Model
AI	Artificial Intelligent
PUT	Program Under Test
MSR	Mining Software Repository
ML	Machine Learning
APR	Automatic Program Repair
MT	Mutation Testing
BERT	Bidirectional Encoder Representations from Transformers
GPT	Generative Pre-trained Transformer
Llama	Large Language Model Meta AI
COT	Chain Of Thought
GP	Genetic Programming
EA	Evolutionary Algorithm
BLEU	Bilingual Evaluation Understudy
CC	Cyclomatic Complexity
AST	Abstract Syntax Tree
IDE	Integrated Development Environment
CR	Correct Ratio
NDC	None Duplicate Correct
NBC	None Duplicate Buggy
RPS	Relative Patch Size
NPC	None Prompted Consideration
MuTAP	Mutation Test case generation using Augmented Prompt
MS	Mutation Score

LIST OF APPENDICES

CHAPTER 1 INTRODUCTION

1.1 Research Context

Numerous techniques and tools have been proposed to assist developers and managers at various stages of the software development process, ranging from the initial requirement engineering phase to the ongoing maintenance. These techniques and tools while attempting to ease the process of development, bear a profound influence on developers' productivity and software quality. However, their effectiveness in enhancing software quality and developer productivity is closely linked to their inherent quality. Assistant tools that deliver low-quality results, such as programming assistant tools generating low-quality code, can erode developers' trust in such tools.

Despite significant efforts invested in methods for developing effective assistant tools for Software Engineering (SE) tasks, there is a notable reluctance among users, including both developers and software project managers, to consistently integrate such tools into their daily workflows [2–4]. This emphasizes the need for an understanding of the limitations of such tools and the proposal of methods that can help improve their effectiveness.

The abundance of data available on various platforms, such as GitHub, and its processing within the domain of Mining Software Repositories (MSR) [5], presents varied opportunities for proposing methods to improve and augment assistant tools in various Software Engineering (SE) tasks. Moreover, the emergence of Machine Learning (ML) and Deep Learning (DL) techniques, notably the recent advancements in generative Artificial Intelligence (AI), holds significant potential for advancing the development of methods that further enhance intelligent assistant tools within the field of SE.

This thesis proposes methodologies that leverage statistical learning to improve the quality and efficiency of intelligent assistant tools across three distinct categories of SE tasks: i) code generation during the implementation stage, ii) test generation within the software testing stage, and, iii) expert identification during development and maintenance stages. While these tasks may not be exclusive to these stages, they find more frequent application in these specific stages [6]. The rationale for concentrating on these tasks stems from their pivotal roles in software development and maintenance as we discuss in the following.

The concept of automatic code generation has long been an aspiration goal in SE research, ultimately aiming to speed up programming activities. Programming assistants aimed to generate code automatically have yielded productivity gains and enhanced developers' effi-

ciency [7, 8]. Developers claim that they are motivated to use programming assistants to accelerate coding tasks, allowing them to shift their focus from keystrokes and repetitive code patterns towards higher-level design considerations and problem-solving tasks [3]. However, to ensure that these tools are used safely and efficiently, it is important to gain a good understanding of their limitations and develop efficient mechanisms to mitigate these limitations.

On the other side, software testing is also a crucial yet expensive step in the software development lifecycle. Software testing can account for 30% to 80% of software development costs [9, 10]. Low-quality test cases can lead to low-quality software. However, generating effective test cases is a time-consuming and often tedious task for developers. Developing high-quality test cases plays a vital role in preventing software bugs. Therefore, developing methodologies and automated assistant tools to support the effective generation of test cases have been a significant focus in SE, intending to reduce the testing burden on developers.

Identifying developers' expertise is also crucial when searching for an individual to hire, to resolve a bug, or to contribute to a software project. The cost of poor recruitment decisions is estimated to be as high as ten times the annual salary of an employee [11]. Large software projects require technical experts across various domains [12]. An accurate assessment of developers' expertise profoundly impacts the quality of software projects [13]. This assessment encompasses a thorough examination of their technical skills, including programming languages [14, 15]. A study by Zhang et al. [16] delves into the factors affecting bug-fixing time and underscores the significance of the delay before bug assignment. One potential cause for this initial delay is the time spent in finding the right developer to tackle the bug. Consequently, the development of a model that can automatically assess developers' programming expertise is a crucial step toward the goal of identifying experts for a specific task in the software development life cycle.

1.2 Problem Statement

Despite significant efforts in developing methods to enhance intelligent assistant tools for code generation, test generation, and expert identification tasks, there remain notable gaps in previous studies.

Code Generation — While the use of LLMs as the core of AI programming assistant tools is promising, challenges persist in ensuring the quality of the generated code [17]. Evaluating that code generated by LLMs is not only syntactically correct but also semantically efficient and aligned with the intended functionality presents a significant challenge. Like human-

written code, code generated by LLMs can introduce bugs and may increase the effort required for maintenance and testing. Previous studies primarily focused on the correctness of code generated by such tools and the types of tasks they can handle, as well as their difficulty levels [18, 19]. However, beyond the correctness, the quality of the code generated by such tools is also important since it can significantly impacts the code contributed by developers to software projects and, consequently, software quality.

Test Generation — Developing methods to enhance intelligent assistant tools for test generation have different drawbacks and often produce tests with either random test inputs, no assertions, overly general assertions, or assertions that cannot effectively assess the intended behavior of the Program Under Test *PUT* [20, 21]. Additionally, developers are often reluctant to use these tools due to the unreadable and non-comprehensive nature of the tests generated by them [2]. Recent studies harnessing LLMs to generate test cases have primarily concentrated on evaluating the efficacy of these test cases in terms of test coverage, neglecting an assessment of their effectiveness in detecting bugs [22]. Nonetheless, previous research indicates that while test cases generated by automatic test generation tools, such as Evosuite [23], may achieve high test coverage, they may not be effective in fault detection [24].

Expert Identification — Previous studies in this category have heavily relied on heuristics such as the *Line 10 Rule* and quantitative metrics like the number of commits in different source files [25–27], or the count of distinct API calls [28, 29] to represent the expertise of developers. While these methods demonstrate good performance in identifying experts within a specific software project, they encounter challenges when employed to represent the expertise of developers across various activities and diverse software projects [30]. In addition, they often lack a solid theoretical foundation and remain susceptible to biases. For instance, as highlighted by Verdi et al. [31], even a highly upvoted code snippet may exhibit low quality and harbor bad practices. Another qualitative study [32] also underscores the significance of "Depth of Knowledge" and "Breadth of General Programming skills" as vital factors contributing to the expertise of developers.

1.3 Thesis Statement

The main objective of this thesis is to propose methodologies that leverage statistical learning to evaluate and enhance the intelligent assistant tools for SE tasks in three key categories: code generation, test generation, and expert identification. This thesis is primarily motivated by the importance of these assistant tools in enhancing software quality and developers' productivity, the existing gaps in state-of-the-art intelligent assistant tools and techniques, and the role and benefits of applying AI techniques to identify and address these gaps.

For tasks that involve code generation, such as code generation and test generation, we leverage the power of LLMs to generate code. The first step before proposing methodologies to enhance the techniques and tools on code/test generation is to comprehend their limitations thoroughly. Hence, our work involves an initial evaluation of the quality and effectiveness of the code/tests generated by LLM-based assistant tools. Subsequently, we present methods to improve the generated code/test and provide recommendations on how to employ these methods to enhance the overall efficiency of the studied coding assistants. Finally, we introduce a taxonomy of bugs that arise in code generated by LLMs, offering guidance for future studies on adapting various SE quality assurance tasks to the specific characteristics of bugs generated by LLMs.

For tasks that consist of assessing the expertise of developers and identifying experts, we employ ML/DL techniques and draw inspiration from theoretical concepts. Recognizing the gap of prior studies in representing and evaluating the expertise of developers across various software projects, we start by introducing a method to represent the domain expertise of developers. Subsequently, we propose a method to assess the expertise of developers based on their contributions to different software projects, as an initial step towards the objective of identifying experts for SE tasks.

1.4 Thesis Overview

1. *Automatic Code Generation.* To highlight the limitations of AI programming assistants and facilitate the development of more effective programming assistant tools, our approach begins with an evaluation of the code quality generated by programming assistants incorporating LLMs, comparing it to human-written code. This evaluation employs various metrics, including the cost of bug fixing. Subsequently, we propose an approach to improve the reliability of code generated by these tools by drawing insights from Automatic Program Repair (APR) efforts, to repair their buggy code and reduce the verification burden on developers. Given that our results indicate variations in bug-fixing efforts between code generated by LLM-based tools and human-written code, in addition, we build a taxonomy of bugs for code generated by LLMs. This taxonomy can serve as a valuable resource for future studies to enhance assistant tools and techniques, such as Mutation testing approaches, to adapt them to the bugs that arise in code generated by LLMs.

2. *Automatic test generation.* To propose a method for enhancing the quality of test cases generated by assistant tools, we begin by evaluating the effectiveness of test cases generated by state-of-the-art tools and LLMs, highlighting their limitations in

revealing bugs. Subsequently, we introduce a methodology to enhance the quality of test cases generated by LLMs through a series of post-processing and prompt-based learning steps. The proposed method improves the quality of test cases generated by LLMs with initial steps in syntax fixing and repairing the intended behavior. It further enhances the effectiveness of test cases generated by LLMs in bug detection by leveraging the potential of Mutation Testing (MT).

3. *Representing the domain expertise of developers.* While previous efforts show different limitations in representing the domain expertise of developers across different projects, we have devised a method that harnesses the "doc2vec" technique [33] to automatically represent the domain expertise of developers with a deeper understanding of semantics in their specific domains across different software projects. We generate embedding vectors using diverse sources of information, extracted from developers' contributions within their GitHub activity. This information spans repository metadata related to their contributions, their bug-resolving history, and the list of APIs applied by developers to various source files. To evaluate the effectiveness of this representation, we undertake a classification task involving the classification of developers into distinct job roles.

4. *Assessing the expertise of developers.* To represent developers' expertise, we consider the mastery of programming language syntax patterns as a proxy for programming expertise. We identify expert developers by evaluating their proficiency in employing a larger and less frequent set of programming syntactic patterns in their code. Our fundamental assumption lies in the fact that a part of the developer's knowledge can be defined by the subset of programming constructs they have mastered. This subset includes elements such as syntactic patterns and lexical expressions within their artifacts. This idea is motivated by a linguistic theory known as Zipf's law [34]. Zipf's law explains the distribution of words within a corpus. The model we propose relies on the Zipf distribution of syntactic patterns present in artifacts created by developers to evaluate their mastery of programming syntax patterns and differentiate between experts and novices.

1.5 Thesis Contribution

This thesis makes the following original contributions:

1. Conduct an empirical evaluation of code generated by Copilot as an intelligent programming assistant tool, in comparison to code written by humans (the first study of

its kind). (Presented in **Chapter 4**)

2. Propose a method to enhance the reliability of code generated by Copilot by leveraging an existing SE tool to repair its buggy suggestions and compare the cost of repair with buggy code written by humans. (Presented in **Chapter 4**)

3. Introduce a taxonomy of bugs in code generated by various LLMs, shedding light on the diverse types and root causes of bugs that may be present in LLM-generated code (the first study of its kind). (Presented in **Chapter 4**)

4. Propose an approach to assess the quality of tests generated by LLMs in detecting bugs using Mutation Testing (MT) (the first study of its kind). (Presented in **Chapter 5**)

5. Propose a novel method that harnesses MT and the potential of surviving mutants to enhance the effectiveness of test cases generated by LLMs in terms of bug detection. (Presented in **Chapter 5**)

6. Propose a method for representing the domain expertise of developers across various projects by applying *doc2vec* to three different sources of their contributions on GitHub, referred to as *"dev2vec"*. (Presented in **Chapter 6**)

7. Propose a novel method to theoretically assess developers' expertise based on the distribution of syntax patterns in their code. (Presented in **Chapter 7**)

1.6 Thesis Organization

The remainder of this dissertation is organized as follows. **Chapter 2** provides background information, while **Chapter 3** provides a comprehensive review of related studies and research work in the field. **Chapter 4** presents an empirical study evaluating the quality of code generated by intelligent programming assistant tools in comparison to human-generated code. Additionally, it introduces a method to improve the reliability of code generated by such tools, followed by a taxonomy of bugs commonly found in code generated by LLMs. **Chapter 5** outlines a method for the automatic generation of test cases using LLMs and introduces approaches for enhancing the effectiveness of the generated test cases through MT. **Chapter 6** introduces a technique for representing the domain expertise of developers in an embedding space using doc2vec. It also includes an evaluation of this representation in comparison to state-of-the-art approaches for assessing the technical specialization of developers. **Chapter 7** offers a theoretically grounded method for assessing developers' expertise across different projects, drawing inspiration from Zipf's law. **Chapter 8** provides a summary and

conclusion of the thesis, while also addressing its limitations and suggesting avenues for future work.

CHAPTER 2 BACKGROUND

2.1 Chapter Overview

In this chapter, we provide essential background materials relevant to this thesis. First, we explain the concepts of LLMs. Subsequently, we introduce various LLMs for code generation. Lastly, we present an overview of Zipf's law and other relevant metrics that we used in this thesis.

2.2 Large Language Model

Large Language Models (LLMs) represent a revolutionary advancement in natural language processing, demonstrating capabilities for general-purpose language understanding and generation [35]. LLMs acquire these remarkable abilities by leveraging massive datasets to learn billions of parameters during training, necessitating substantial computational resources for both training and operational phases [36].

These models are artificial neural networks, with the *transformer* architecture [37], such as BERT (Bidirectional Encoder Representations from Transformers) [38] which are (pre-)trained using self-supervised learning and semi-supervised learning. The transformer architecture, characterized by self-attention mechanisms [37] serves as the fundamental building block for language modeling tasks.

The autoregressive nature of LLMs involves taking an input and predicting the subsequent word or token in a sequence. This approach has demonstrated effectiveness across broad applications, ranging from language translation to code generation. Notable examples of these models include OpenAI's GPT [39] series (including GPT-3.5 and GPT-4, utilized in ChatGPT), Google's PaLM [40] (deployed in Bard), and Meta's LLaMa [41].

Pre-training: Pre-training in LLMs denotes the initial training phase, encompassing both self-supervised learning, where the model predicts missing words or sequences in unlabeled data, and semi-supervised learning, integrating labeled data to fine-tune the model for specific tasks. The term "pre-training" is employed because it anticipates the need for additional training or post-processing steps to adapt the pre-trained model to the desired task [35].

Fine-tuning: Fine-tuning involves taking pre-trained models and refining them through additional training on smaller, task-specific labeled datasets [35]. This process adapts the models' capabilities and enhances their performance for a particular task or domain. Essen-

tially, fine-tuning transforms general-purpose models into specialized ones. An example of such a task is fine-tuning CodeBERT for defect detection task [42].

In-context Learning: Large Language Models (LLMs) demonstrate an ability for in-context learning, meaning they can learn effectively from a few examples within a specific context. The fundamental concept of in-context learning revolves around the model's capacity to learn the patterns through the examples and subsequently make accurate predictions [43]. One advantage of in-context learning is the possibility of engaging in a dialogue with the model. Secondly, in-context learning closely aligns with the decision-making processes observed in humans by learning from analogy [44]. In contrast to traditional training and tuning approaches, in-context learning operates as a training-free framework, significantly reducing the computational costs associated with adapting the model to new tasks. Moreover, this approach transforms LLMs into black-boxes as a service that can be integrated into real-world tasks [45]. Different methods have been proposed in this category, including few-shot learning [39], Chain of Thought (COT) [46], and self-planning [47]. In few-shot learning, diverse task-representative examples, as input/output pairs are incorporated into the prompt. The COT is motivated by the natural step-by-step thinking ability of humans and it improves the performance of LLMs in solving problems that involve multi-step reasoning. The self-planning approach involves breaking down original tasks into smaller steps using LLMs and calling them plans and the call the model on the provided steps.

2.3 Large Language Models for code generation

Various LLMs exhibit remarkable performance in the realm of code generation in SE. Table 2.1 shows a list of well-known LLMs for code generation tasks, along with their prevalent use cases. An early undertaking on pre-training transformers with code content is CodeBERT [48]. CodeBERT is an auto-encoding LLM based on BERT architecture [38]. Its pre-training phase encompasses 6M (Million) code snippets and 2M pairs of natural language and code snippets, spanned across 6 programming languages. Its pre-training data are all extracted from the CodeSearchNet [49] dataset gathered from GitHub public repositories. It has found application in diverse code classification tasks such as code review classification [50], predicting flaky tests [51], defect detection [42] , code clone detection [52] and also in code retrieval tasks [53]. While CodeBERT continues to stand out as a practical, open-source, and lightweight model, well-suited for a variety of tasks that require code representation [54,55], its maximum length of tokens limits its usage for diverse code generation tasks.

One of the very powerful LLMs is OpenAI's Codex [56], extensively applied across various

SE tasks involving code generation [22,57]. Codex is a GPT-based auto-regressive LLM [39]. with up to 12B (Billion) parameters that is fine-tuned on 54M public repositories from GitHub. OpenAI provides two versions of Codex: *code-davinci-002* and *code-cushman-001*. The former stands as the more capable model, with a maximum context length of 4k, whereas the latter is limited to 2k. Notably, Codex powers GitHub Copilot[1], an in-IDE developer coding assistant that was released on June 29, 2021 and is adept at generating code in response to the context provided by the user.

A new release of GitHub Copilot is "Copilot Chat"[2] which is tuned with human feedback for dialog use cases. Copilot Chat can be employed for a wide range of coding-related tasks. Depending on the specific context of the task, it offers responses such as generating code snippets to implement specific functionalities, providing natural language explanations for code segments, suggesting unit tests, or repairing a buggy code. Although Codex and its collaborative tool, Copilot, demonstrate promising performance across a range of code generation tasks, their lack of open-source availability renders them unsuitable for tasks that involve the tuning step.

CodeT5 operates as a sequence-to-sequence LLM [58]. CodeT5 employed code-specific knowledge and proposed a novel type identifier-aware approach as part of its training objective. It is built upon the T5 architecture and is well-suited for tasks revolving around the generation of a new sequence in response to an input sequence. An illustrative example of such tasks including code translation [59], code summarization [60], or generative question answering [61]. It learns the representation of identifiers and their respective type by masking them during the training since usually in code developers use identifiers in a way that saves a lot of semantics about the functionality of the code [58]. The model is initially trained on CodeSearchNet [49] data and then refined on various tasks within CodeXGLUE [62], which includes tasks like detecting code defects, identifying clones, and performing generative tasks such as code summarization, translation, and refinement. The updated version of CodeT5, called CodeT5+, has been trained on a larger volume of data, encompassing 115M code files sourced from GitHub [3] and it is configurable to decode-only or encode-only architecture as well. Although CodeT5 is an open-source model that makes it suitable for studies that require fine-tuning, it cannot compete with Codex in Code generation tasks.

Llama-2 is a powerful and open-source LLM that has been introduced by Meta. It is an auto-regressive model, available from 7B to 70B parameters [41]. This auto-regressive model is pre-trained on a mix of publicly available data, ending up to 2T (Trillion) tokens. The fine-tuned

[1] `https://copilot.github.com/`

[2] `https://docs.github.com/en/copilot/github-copilot-chat`

[3] `https://huggingface.co/datasets/codeparrot/github-code`

Table 2.1 Common LLMs for SE tasks involving code generation.

Model	Architecture	Transformer Type	Parameter Size	Max Length	Common Use Case
CodeBERT	BERT	Encoder-only (auto-encoding)	125M	512	Classification tasks: code vulnerability classification Information Retrieval tasks: code search
Codex	GPT	Decoder-only (auto-regressive)	12M-12B	4K	Generation Task (left-to-right): code completion and code generation
CodeT5	T5	Encoder-Decoder (sequence-to-sequence)	60M, 220M and 770M	1K	Code Translation, Code Summarization, question/answering
Llama-2-chat	Llama	Decoder-only (auto-regressive)	7B-70B	4K	Dialog use case on variety of generative tasks including code generation, code summarization, code translation

version of llama-2, known as llama-2-chat, is iteratively tuned using Reinforcement Learning with Human Feedback (RLHF). This process optimizes the model specifically for dialog use cases, spanning different types of tasks, including programming [41]. The dialog model gives the possibility of defining three roles: *System, User*, and *Assistant*. The system role helps to specify the task type or the programming language that meets the user's intention.

2.4 Zipf's law

This section provides a brief background on Zipf's law, which is a fundamental concept underpinning the proposed approach in Chapter 7.

Human language is characterized by various patterns, structures, and grammatical rules. The arrangement of words, symbols, and spaces in language adheres to hidden rules. Researchers have discovered different statistical laws in linguistics. One well-known example is Zipf's law, first discovered by George Zipf [34]. Zipf's law is grounded in the concept of "least effort", which governs the transmission of information from speaker to listener [34]. This principle asserts that people tend to solve problems in a manner that minimizes their effort [63].

Zipf's law argues that when words in a text are sorted by their frequencies, the frequency of each word exhibits an inverse relationship with its rank. Equation 2.1 demonstrates this relationship, where f_w and r_w denote the frequency and rank of word w and C represents a constant [34, 64].

$$f_w = C/r_w^{\alpha} \tag{2.1}$$

Equation (2.2) shows the same relationship but using probability of words instead of their frequency, where $Pr(r_w; \alpha, n)$ represents the probability of a word w, r_w denotes its rank, n stands for the number of distinct words, and α is an exponent that typically approximates to 1 [65]. It is worth noting that a probability distribution can exhibit $\alpha < 1$ under certain circumstances, particularly when n is constrained by an upper limit [66]. The term H_n

represents a Harmonic series utilized for normalization in the equation.

$$Pr(r_w; \alpha, n) = \frac{1}{r_w^{\alpha} * H_{n,\alpha}} \tag{2.2}$$

If we apply a logarithm to both sides of (2.2), we obtain (2.3), which is a linear representation of Zipf's law. This transformation converts a power-law function into a straight line when plotted in a log-log space, with an intercept of $\log H_{n,\alpha}$ and a slope of α.

$$\log Pr(r_w) = -\alpha \log r_w - \log H_{n,\alpha} \tag{2.3}$$

Zipf's law has been observed in both natural language contexts such as English corpus [67] and also programming language source files such as Java and C++ [65].

2.5 Automatic Program Repair

One of the common techniques in automatic program repair (APR) is known as the generate-and-validate approach [68]. In generate-and-validate approaches, a predefined search space is explored to generate a candidate patch until the candidate patch successfully passes all the test cases. Several studies have utilized correct programs to construct the search space and derived the minimum-size patch required to repair the buggy program. An example tool that leverages such an approach in Python is "Refactory" [69]. "Refactory" employs dynamic analysis and introduces a fully automated method for generating repairs in introductory programming tasks. For a buggy program, it first compares it to a refactored correct program based on its control-flow structure. Then, the method infers the input-output specifications for the buggy program from the executions of the correct program. Finally, it employs the inferred specifications to modify the blocks of the buggy program through search-based synthesis. Notably, this tool demonstrates better performance compared to other state-of-the-art APR tools like Clara [70]. Despite other techniques that need a large and diverse range of correct programs, "Refactory" can repair the buggy program even with one or two references (i.e., correct code).

Given that in code snippets generated by Copilot for a programming task, we often encounter one or more correct suggestions (such as students' submissions), and each programming task in our dataset has a corresponding current solution, we leverage "Refactory" as a suitable tool to repair buggy code generated by both Copilot and students. This tool offers an additional advantage by providing useful metrics for reporting repair costs, enabling us to compare the quality of code generated by Copilot and humans in terms of repair cost. Following our study,

other research endeavors have also utilized this tool to repair the buggy code of LLMs or to employ it as a baseline tool for evaluating the potential of LLMs as APR tools [71, 72].

2.6 BLEU Score

The BLEU (Bilingual Evaluation Understudy) score quantifies the similarity between a machine-generated text or code (referred to as a hypothesis) and one or more ground truths. Equation 2.4 demonstrates how the BLEU score is computed. It takes into account precision, which measures the number of overlapping n-grams between the tokens of the hypothesis and the ground truth, as well as brevity, which penalizes differences in length. The BLEU score yields a value between 0 and 1, with higher scores indicating greater similarity [73, 74].

$$BLEU\_score = BP.e^{\frac{1}{n}.\{log(P1)+...+log(P_n)\}} \tag{2.4}$$

In Equation 2.4, BP represents the brevity penalty, which penalizes the hypothesis context if its length is significantly shorter than the closest match in the ground truths. To calculate P_i, we assess the overlap between the bag of n-grams in the hypothesis context and the ground truth, as depicted in Equation 2.5. If we denote S^i_{ht} as the set of n-grams in the hypothesis and S^i_{gt} as the set of n-grams in ground truth, then P_i is measured as follows:

$$P_i = \frac{|S^i_{ht} \cap S^i_{gt}|}{S^i_{ht}} \tag{2.5}$$

While BLEU is a common metric for evaluating text and code generated by LLMs, it may not always align perfectly with human judgment. This is because it primarily relies on n-gram overlap, which, in the case of text, does not account for factors like fluency, grammar, or meaning, and in the case of code, it does not consider functional correctness. Nevertheless, it remains widely used for LLM evaluation [75]. To address this limitation in evaluating code, a metric called CodeBLEU measures the BLEU score in terms of code syntax and semantics. As part of this metric, CodeBLEU computes the BLEU score between the AST of two code snippets [74]. While the BLEU score is commonly used to evaluate machine-generated text quality, particularly in machine translation tasks, it also finds applications in assessing program synthesis approaches such as text-to-code, code summarization, and code prediction [73, 74].

2.7 Cyclomatic Complexity (CC)

Cyclomatic Complexity (McCabe's Cyclomatic Complexity or CC) serves as another code quality metric that assesses the understandability of a code snippet. CC quantifies the number of independent paths in a code component, specifically, the number of decision points within the source code [76, 77]. Measuring code snippet understandability enables us to estimate the effort required for adding new features to the code or making modifications [78].

Several studies have employed CC to evaluate the readability and understandability of small code snippets [17,79,80]. When comparing solutions for a problem, a lower CC value signifies more readable and understandable code.

CHAPTER 3 LITERATURE REVIEW

3.1 Chapter overview

In this chapter, we delve into the landscape of literature encompassing multiple aspects of automation in SE tasks as assistant tools, specifically focusing on different aspects of code generation, test generation, and program repair using LLMs. The chapter also delves into the concept of developers' expertise, discussing various expertise models, the approaches and challenges to evaluate the effectiveness of these models, and the methods to represent the domain expertise of developers. We explore the intriguing phenomenon of Zipf's law and its presence in both natural language and programming languages. Additionally, we investigate the use of embedding vectors within the software domain.

3.2 Automatic Code Generation

Code generation involves the transformation of an intended logic into executable programs [81]. There is a long history of efforts for automatic code generation. Previous research used several heuristics extracted by humans for automatic code generation [82]. Early efforts in this field have modeled centered around an assistant tool that framed code generation tasks as a search problem, where the goal was to discover a program within a predefined search space of a programming language that satisfied all constraints derived from task specifications [83]. Subsequent efforts shifted towards input/output task specifications, involving the collection of a set of expressions from the pool of possible expressions in a programming language that transforms input examples into output examples [84].

The advent of generative AI has brought a profound transformation to the conception of developing intelligent assistant tools for automatic code generation assistants. Various initiatives have concentrated on leveraging LLM to generate code that aligns with a natural language description of a programming task, called prompt [8, 85, 86].

There are studies that empirically investigate the diverse capabilities of AI programming assistant tools incorporating LLMs like Copilot. Sobania et al. [87] compared Copilot with a Genetic Programming (GP)-based approach that achieved high performance in program synthesis. Their findings reveal that GP-based approaches require more time to generate a solution. Furthermore, training GP-based models is expensive due to the high cost of data labeling. Additionally, these approaches are unsuitable to support developers in practice since GP typically generates code that is bloated and challenging for humans to comprehend.

Vaithilingam et al. [88] conducted a human study involving 24 participants to gain insight into how Copilot can assist programmers in completing tasks. They specifically focused on three Python programming tasks: 1) editing CSV files, 2) web scraping, and 3) graph plotting. Their findings indicate that while Copilot did not necessarily lead to improvements in task completion time and success rates, programmers prefer to incorporate Copilot into their daily tasks because it provides valuable starting points for addressing the tasks. It's worth noting that the tasks in this study required less problem-solving effort compared to the typical programming tasks in our first study (Chapter 4), as they mainly involved the use of programming language libraries. Additionally, the study did not involve a comparison of Copilot's suggestions with those made by the participants when they worked without the assistance of Copilot.

Drori and Verma [18] conducted a study to assess Copilot's ability to solve linear algebra problems within the context of the MIT linear algebra course. In a similar line of work, Tang et al. [89] explored Copilot's capability to solve university-level probability and statistical problems. Both of these studies primarily concentrated on evaluating the correctness ratio of Copilot's solutions and did not delve into an examination of the quality of the code it suggested.

Finnie-Ansley et al. [19] employed Davinci(a version of Codex), in their study using two different datasets. The first dataset comprised 23 programming questions from a programming course, students' solutions to these questions, and their corresponding grades. This dataset is not publicly accessible. The second dataset consisted of various descriptions of a well-known problem, namely rainfall, without any accompanying human-generated solutions. For the programming questions, the paper's focus centered on grading the solutions provided by Codex. This involved generating the correct solutions for the problems across multiple runs (specifically, 10 runs) and subsequently comparing these results with the grades of the students. In the case of the second dataset, apart from assessing code correctness, the authors also examined the diversity of solutions by calculating the number of source lines of code (SLOC). Their findings indicated that Codex outperformed most students, as evidenced by the grades received for its proposed solutions. Furthermore, they observed that utilizing the same input as a prompt on Codex could yield different solutions. This variability may stem from the decoding mechanism in Codex, which involves sampling tokens from a probability distribution after applying the temperature [39, 56].

Nguyen and Nadi [17] evaluated Copilot on 33 LeetCode questions in 4 different programming languages. They used the LeetCode platform to test the correctness of Copilot's solutions. The questions in their study included different levels of difficulty. Although they evaluated

the correctness of Copilot's solutions and compared their understandability.

Another set of studies delves into vulnerability-related issues in evaluating Copilot's solutions. As previously mentioned, Copilot is trained on a substantial volume of publicly available code repositories on GitHub, which may contain bugs or security vulnerabilities. Pearce et al. [90] conducted various scenarios focusing on high-risk cybersecurity problems. Their aim was to investigate whether Copilot has the capacity to learn from flawed code and generate insecure code. In another study, Asare et al. [91] explored how Copilot can replicate vulnerabilities present in human-generated programs. To accomplish this, they initially employed a dataset of vulnerabilities created by humans. They then reconstructed the entire code, excluding the bug, and tasked Copilot with completing the code. The completed section was subjected to manual inspection by three coders to determine whether Copilot reproduced the vulnerability or fixed it.

Moroz et al. [92] examined the challenges and the potential of Copilot to improve the productivity of developers. They highlighted the copyright problems and the safety issues of its solutions. They discussed the non-deterministic nature of such models and the harmful content that could be generated by them. Authors in [93] conducted a survey involving 2631 developers to assess the impact of Copilot on their productivity. They also emphasized various metrics of users' interaction with Copilot that can be used to predict their productivity. To generate these metrics, the authors utilized the SPACE framework [94], which enabled them to formulate 11 Likert-style questions for their survey. Additionally, they analyzed the usage data of Copilot from the participants who responded to the survey. From this data, they extracted various metrics, including the acceptance rate of solutions, persistence rate, unchanged and mostly unchanged accepted solutions, and more. Their findings indicated that the acceptance rate of solutions by developers stands out as the most relevant metric for assessing the impact of Copilot on developers' productivity.

In another study, the author discussed the opportunities and challenges of AI code generation tools in an educational context [95]. This study is not an empirical study and there is no experiment or evaluation of Copilot's suggestions. The author was not specifically focusing on Copilot and its competence or limitations, but on the opportunities/challenges that such tools (code generation tools) bring into education such as exercise generation, illustrative samples, example solutions, etc.

The authors in [96] evaluated Copilot's capabilities in solving elementary coding problems that are taught to students in an introductory programming course (CS1: Introduction to Programming) and investigated the effect of modifying the task descriptions, also known as prompt engineering, to address the unsuccessful attempts at solving the tasks using the

original descriptions. They also categorized the areas of failure on those programming tasks. The authors explored the impact of using different natural language descriptions in decreasing the number of failures or wrong suggestions. Their focus was only on the correctness (correct/fail) of Copilot's suggestions, not their quality if used as an AI pair programmer. They did not conduct a comparison between Copilot's suggestions and those generated by students.

In [97], the author studied Copilot's capabilities on a handful of programming tasks designed to test students' knowledge of program design. The author also compared Copilot's outputs to those of Codex and assessed the correctness of their suggestions alongside the diversity of their solutions. For diversity comparison, the author manually checked the different programming structures used in those solutions. The author also assessed Copilot's and Codex's abilities in generating test cases while explaining their own solutions.

Authors in [98] studied how Codex can be used to provide better programming error messages. The authors provided a faulty program alongside the prompt asking the model to explain why the program fails and to provide solutions. They analyzed the provided solutions in terms of whether the solution is correct, understandable, contains a fix to the fault, and whether it improves upon the original script.

In addition to the model's performance, which can affect the correctness of the generated code by such tools, another correlated factor that has been identified is the complexity or ambiguity of the prompt [99, 100]. Various approaches have been employed to address this limitation, including few-shot learning [39] and self-planning [47].

The experiences of developers in using tools such as Copilot or ChatGPT demonstrate that these tools assist them in quickly completing programming tasks [7, 101]. Other studies show these tools save developers' efforts in online searching and offer a good starting point for implementing task specifications [88, 102].

Although LLM-based programming assistant tools provide numerous insights into completing programming tasks, the results of all these studies prove that the suggestions of these tools can occasionally be plagued by bugs, non-compilable code, or misalignment with task specifications [88]. Consequently, in addition to a set of test cases used to evaluate the correctness of suggestions, various tools, and methods are also necessary to filter out or repair low-quality suggestions. This is the concentration of our study in Chapter 4.

3.3 Automatic Test Generation

Unit tests are essential as they serve as the foundation of the test automation pyramid [103, 104]. Unit tests verify whether a function or component behaves as expected in isolation. A unit test comprises two key components: the first component consists of test inputs for the Program Under Test (PUT), while the second component serves as the test oracle, which defines the intended behavior (i.e., output) of the PUT and is capable of uncovering bugs by verifying the correctness of the PUT when subjected to test inputs [105]. Test oracles often take the form of assertions. Incorrect or incomplete test oracles can result in inaccurate test results.

Diverse tools are available for the automated generation of unit tests and test suites, employing various methodologies such as random test generation [106, 107], dynamic symbolic execution [108, 109], and search-based approaches [23, 110]. Notably, Evosuite [23] for Java and Pynguin [111] for Python have attracted significant attention within the research studies. EvoSuite, in particular, has been instrumental in popularizing the use of Evolutionary Algorithms (EAs) for test suite generation [112]. Both Evosuite and Pynguin follow a similar search-based approach, starting with a randomly generated test suite and iteratively mutating it. They preserve mutated test suites that exhibit greater test coverage than others. Mutations at the test suite level include altering (adding, removing, or inserting) different variables and statements within test cases.

Following LLM's good performance in code synthesis, another realm of code-related tasks that have gained from the emergence of LLM is software testing. Automatic test generation has been a longstanding research goal within the field of SE. Testing constitutes an important yet costly phase in the SE development lifecycle.

Authors in [113] studied the impact of *few-shot* learning across various downstream tasks, including test case and test oracle generation. They compared the performance of *few-shot* learning with automatic test generation tools. The investigation was conducted on a different set of Java methods sourced from different benchmarks. The outcomes indicated that LLMs possess the capability to generate test cases and test oracles that exactly match (in lexical terms) the ground truth tests within the benchmark projects. Furthermore, their test coverage was found to be comparable with test cases generated by automatic test generation tools.

Schafer et al. [114] undertook an effort to generate test cases by prompting Codex. Their investigation was concentrated on 25 JavaScript packages. The prompt in their study encompassed the implementation of the PUT and also the usage examples of APIs extracted

from documentation. In instances where a test case proved unsuccessful on the PUT, their method incorporated the encountered error message into the prompt and re-prompted Codex. Their findings demonstrated that the process of enhancing the prompt with such additional information facilitated Codex in producing correct test cases with sufficient coverage.

LIBRO [115] used the issue reports (both title and body) as *few-shot* prompts to generate bug-reproducing test cases. The final test cases were incorporated into appropriate test classes and ranked based on their validity. The results revealed an enhancement in generating correct test cases to reproduce bugs compared to state-of-the-art tools.

CEDAR [116], rather than employing fixed demonstrative examples in *few-shot* learning, aimed to retrieve demonstrative examples related to each *PUT* and incorporate them into the prompt. They assessed their method based on the lexical match, termed "exact match", between generated assertions and the ground truth in a benchmark. While their proposed approach demonstrates enhanced performance in achieving exact matches between assertions and the ground truth, it necessitates an extensive pull of code samples for the selection of appropriate demonstrative examples for each *PUT*.

ATHENATEST [117] employed the BART transformer model [118], which they fine-tuned using a collection of Java functions and their corresponding tests. They reported test coverage comparable to those of EvoSuite [23] upon evaluating generating test cases for five Java projects.

TOGA [55] engaged in fine-tuning CodeBERT using the *PUT*'s docstring along with the prefix of a test case featuring a masked assertion. Their goal was to synthesize the assertion. Subsequently, they formulated the whole test oracles by incorporating a test oracle grammar and generating a set of assertions. This set was then subjected to ranking through a neural network ranker based on their lexical match to ground truth test oracles. Although they reported results akin to those of EvoSuite [23] in bug detection, their focus is only on synthesizing the assertions. However, synthesizing assertion is not challenging but generating effective and meaningful test oracles poses a significant challenge.

CODAMOSA combined the test cases generated by Codex with those derived from Pynguin in cases where Pynguin's test case generation halted and failed to enhance test coverage. CODAMOSA achieves higher test coverage on various Python benchmarks [22] compared to Pynguin. It is worth noting that, akin to other studies, CODAMOSA concentrated solely on test coverage improvement, and its generated test cases lacked assertion oracles for bug detection within programs.

Two additional studies employed Codex to simultaneously generate code and corresponding

test cases based on a given problem description. Subsequently, they used these test cases to filter out buggy suggestions produced by Codex [119, 120]. For code generation, they employed the problem description as a prompt, and for test case generation, they used the same problem description along with the PUT and a natural language instruction.

Although prior research has explored diverse strategies for generating test cases using LLMs like Codex and assessed them in terms of test coverage or lexical match with ground truth tests, none of these studies specifically focused on leveraging MT to enhance the effectiveness of the generated test cases.

3.4 Automatic Program Repair

Another domain within the SE development life cycle that has been influenced by LLM is program repair. Automatic program repairs aid developers in enhancing code reliability. Various LLMs such as CodeT5 [121, 122], Codex [123], or ChatGPT [124] have been employed for programming repairs. Their results show that directly applying LLMs for automatic program repairs outperformed different state-of-the-art tools across various levels of granularity: producing complete patches, inline repairs with provided prefixes and suffixes for buggy functions, and repairing individual lines [121].

The adoption of retrieval few-shot learning, which exhibits improvements in test case generation, also yields interesting outcomes in the program repair [125,126]. Fine-tuning pre-trained LLM with bug-repair datasets is another effort in this area [122]. While fine-tuning shows some improvement in program repair compared to direct LLM prompting and state-of-the-art program repair tools, it is expensive and demands a considerable volume of data. LLMs are also employed for repairing bugs in both programs and test cases that have been generated by LLMs themselves [114, 127]. This is accomplished by re-prompting the LLMs with for example the inclusion of error details in the prompt.

3.5 The expertise of developer

Previous research aimed at assessing programming expertise can be categorized into two general groups: *"Internal expertise"*, which focuses on identifying experts within a software project, and *"Overall expertise"*, which assesses developers' expertise based on their contributions to various software projects. Research pertaining to *"Internal expertise"* of developers, also known as *"Implementation expertise"* [27], seeks to identify experts capable of performing specific tasks within a software project.

On the other hand, *"Overall expertise"* studies concentrate on evaluating developers' expertise by analyzing their contributions across different software projects. The importance of this group can be traced to the efforts of open-source project managers who rely on information about developers' contributions to projects hosted on platforms such as GitHub to identify potential contributors. Despite a wide range of research falling into the first category, there is a limited number of studies in the second category.

Both categories of expertise models face two main challenges: representing developers' expertise and assessing the effectiveness of the representation models. In the following sections, we will delve into how previous research has addressed these challenges.

3.5.1 Expertise model of developers

The initial step in constructing an expertise model for programming is the representation of developers' expertise. The majority of prior research in both expertise categories constructs their models for assessing expertise based on various features, such as the number of commits, votes, or changes in a file path. In defining internal experts, a commonly adopted heuristic is the *"Line 10 Rule"* [25, 26], applied in different contexts. This rule draws inspiration from version control systems, which store the author's name in line 10 of the commit log, signifying that if a developer has previously modified a file, they are considered as a potential candidate for addressing tasks related to that file. These methods cannot be expanded to model the expertise of developers across different projects because they are limited to the path of a source file.

For example, authors in [128] consider the number of changes made by developers in the history of various files as a measure of their expertise. In other studies, such as [129] and [130], researchers examine the locations of files and compile lists of experts based on the history of file paths and the names of authors. In the work presented in [27], a network of experts is established based on the names of developers found in the carbon-copy of bug reports, and recommendations are provided as a list of developers capable of resolving bugs.

Studies seeking to identify experts in Question Answering (QA) or Crowdsourcing platforms often rely on similar straightforward features. For instance, authors in [131] calculate the number of questions and answers to represent an individual's knowledge in QA platforms. Other studies, such as [132] and [133], leverage the number of votes on different answers by an individual to estimate their expertise. In Crowdsourcing platforms, the number of completed tasks is collected to gauge expertise, as demonstrated in [134].

In an alternative approach, as seen in the study detailed in [135], the contribution of devel-

opers on platforms like GitHub and StackOverflow is combined, and the tags associated with questions or tasks completed by an individual are used to represent their knowledge.

Other studies go further, utilizing linguistic information within code or API calls to represent developers' knowledge. Certain studies gather comments, variable, or function names from 'diff' files created by specific developers to define their expertise [14, 136]. In the realm of Question Answering (QA) platforms, one study focuses on the textual descriptions of questions and answers to detect the expertise of members, calculating the similarity between new question descriptions and previous ones to identify experts capable of providing answers [137].

In addition, authors in [28] and [138] consider the number of distinct API calls as indicative of developers' domain expertise. Within QA platforms, a study [139] collects function and method names from questions and answers to represent the knowledge of contributors. Another research [140] compiles all tags from Stack Overflow as technical keywords. Instead of considering all linguistic information within bug report descriptions, they gather Stack Overflow tags from bug descriptions to rank developers based on their expertise in addressing new bugs.

Heuristic approaches that gather quantitative features, such as the number of commits, votes, API calls, or textual descriptions like comments and identifiers, can occasionally serve as effective indicators of developers' knowledge. However, these approaches often lack a strong theoretical foundation and may be susceptible to biases. For instance, as highlighted by Verdi et al. [31], even a highly upvoted code snippet may exhibit low quality and contain vulnerabilities. Thus, a high voted response cannot be a good indicator of expertise.

A survey study [141] involving 1,926 software engineers was conducted to collect top features that distinguished great software engineers, revealing that the ability to write high-quality code is one of the primary characteristics of good software developers. Another qualitative study [32], underscores the significance of "Depth of Knowledge" and "Breadth of General Programming" as vital factors contributing to the expertise of developers, in addition to their soft skills.

However, prior investigations typically did not delve into the content of the code itself. In our research, outlined in Chapter 7, we introduce a more theoretically grounded approach to model developers' expertise, primarily based on the content of the source code they produced. Our emphasis is on programming mastery, and we do not factor in the soft skills of developers when evaluating expertise.

3.5.2 Assesing the effectiveness of the expertise model

To validate the effectiveness of models in representing developers' expertise, studies focusing on *Internal Expertise* use bug resolving history of developers [142] and task completion history [143] to evaluate the performance of their models. This approach works well for developers with an established activity history. However, it poses challenges for developers with little or no record of past activities within the projects because these methods rely on developers' activity to estimate their expertise. Consequently, developers lacking activity records might not be identified in the suggested developer lists.

The evaluation of *Overall Expertise* of developers is even more challenging due to the absence of a concrete ground truth. One research effort [15], conducts a qualitative study to generate such ground truth. They administer a self-evaluation survey, asking developers to assess their expertise in three well-known Java libraries, using a rating scale from 1 to 5. The results of these self-assessments reveal a tendency for overestimation or underestimation in certain categories. In response, one of the authors performs manual searches on LinkedIn to verify the expertise of a group of developers. Another set of studies seeks the evaluation of external experts, such as groups of students in the relevant field or project managers, to assess the outcomes of their models [138, 144]. CVExplorer [145] validates its results by recommending job candidates for positions in two companies.

We contend that the lack of ground truth and the primary challenge of evaluating models representing developers' expertise across different projects is the main reason for the limited number of studies dedicated to assessing *Overall Expertise* of developers.

3.5.3 Domain expertise of developers

Numerous studies in the area of developer expertise models have concentrated on the task of finding the topic of source code and identifying experts in specific domains, as expertise is topic sensitive. studies such as [146] and [147] explored how to uncover different topics within large software systems by employing Latent Dirichlet Allocation (LDA) on selected contexts of code. For instance, keywords like "SQL", "authentication", and "db" often occur in code snippets related to the database domain.

A noteworthy example is $Topic_{XP}$ [148], an Eclipse plugin that employs LDA to cluster classes and libraries within a Java project into various topics. In a similar vein, authors in [149] harness linguistic information within source code, including variable names, using Latent Semantic Indexing techniques to reveal the code's intention and cluster source code artifacts into different topics. Developers who have authored artifacts related to a specific topic are

subsequently recognized as experts in that domain. However, the authors acknowledge the influence of arbitrary names on the accuracy of their model.

DRETOM [150] is a developer recommendation system for resolving bugs that rely on topic modeling of developers' bug resolution histories. It takes into account the textual descriptions of bug reports and categorizes these reports into distinct topics. This system establishes connections between developers who have resolved specific bugs and the topic associated with those bugs to identify the developers' areas of interest or domains of expertise. For new bug reports, it ranks developers based on their domain of expertise and the relevant topic of the bugs. It is essential to note that all these approaches primarily focus on predicting developers' domain expertise within a single software project.

While these methods excel in defining the domain expertise of developers within a single software project, collecting such information across various software projects introduces additional features, which, in turn, amplifies sparsity and impacts the performance of these methods. Montandon et al. in [30] represent developers' domain expertise by aggregating their activities across various projects. They employ the bag-of-words technique to analyze data sourced from developers' Github biographies, repository descriptions, programming languages, and project dependencies as a means of representing their expertise. To mitigate sparsity, they reduce the dimensionality of the collected data by calculating correlations between various features and selecting those with high correlations. We replicated this study as a relevant state-of-the-art approach to compare it with our research in Chapter 6.

3.6 Zipf's law in human and programming language

Zipf's law, first observed in English text, has subsequently been identified in various languages, including German, Chinese, and Russian [67]. Notably, some studies have detected Zipf's law in programming languages as well. For instance, in [65], the author establishes that the distribution of lexical tokens in Java, C++, and C source code adheres to a Zipfian distribution. Another study, as presented in [151], illustrates that the fat-tailed distribution of function calls and classes in software aligns more closely with Zipf's law than with lognormal or stretched exponential distributions. In the domain of software engineering, research conducted in [152] applies the Pareto and Zipf law principles to software defects, revealing that a mere 20% of source files are responsible for 80% of defects.

Some studies undertake a comparative analysis of Zipf's law distribution in human language and programming language, such as [153] and [154]. They find that the range of parameters in Zipf's law differs between humans and programming languages. The challenge of parameter

fitting is a central aspect of Power law functions, and several studies have explored various methods for parameter fitting and assessing goodness of fit [155–157]. They contend that fitting Power laws through a least-squares approach based on a linear regression model in a log-log space introduces bias, advocating the use of maximum likelihood for parameter estimation and the Kolmogorov-Smirnov (KS) test to assess goodness of fit [158].

A psychologically grounded study in linguistics has uncovered a connection between the parameters of Zipf's law and linguistic complexity [159]. The study reveals that the parameters of Zipf's law in children's speech exhibit different ranges and fluctuations compared to those in adult speech.

In Chapter 7, we will explore the Zipf distribution, employing it to represent the distribution of syntax patterns in developers' artifacts. This will enable us to distinguish between experts and novices using a parameter within Zipf's law.

3.7 Embedding vectors in Software Domain

In SE, vector embedding methods have gained widespread popularity for a variety of purposes. Zhang et al. [160] employ average word embedding and document embedding for issue knowledge acquisition. They generate embedding vectors for each issue by analyzing the content in their titles and bodies. Subsequently, they establish connections between issues and potentially related ones by computing the similarity between issue vectors.

Code2vec [161] transforms code into embedding vectors through model training using different paths obtained from the Abstract Syntax Tree (AST) of methods, with the aim of predicting method names for code snippets in the test set. Another study applies the pretrained code2vec model to various commits, representing them as embedding vectors, and then classifies commit vectors as either security-relevant or non-relevant [162].

In another study, API calls in the source files of three programming languages (Java, JavaScript, and Python) were represented as embedding vectors. To evaluate the performance of these vectors in representing API calls, all APIs within a source file were considered as related. The study found that the vectors of these APIs exhibited greater similarity than those of APIs that never appeared together in a source file [163].

Dey et al. [164] collected API calls from developers' commits across various projects in 17 programming languages. They acquired embedding vectors for three distinct entities: APIs, projects, and developers, collectively termed the "API-related Skill Space" of projects and developers. They observed that developers were more inclined to utilize new APIs or engage in new projects with similar representations within the API-related Skill Space. However,

they did not assess their model's ability to predict the embedding vector for a new developer who was not encountered during the training phase.

In Chapter 6, we employ a similar embedding technique to represent developers' domain expertise as demonstrated by their contributions to various projects.

3.8 Chapter Summary

This chapter offers an exploration of the literature focusing on the automation of various SE tasks, with a particular emphasis on automatic code generation, test generation, program repair, and developer expertise. It delves into the utilization of LLMs for automating tasks that require code generation, highlighting their potential and applications. However, a noteworthy gap in the existing research is revealed, as there is a limited investigation into the assessment and enhancement of the quality of generated code, particularly concerning diverse quality metrics. Moreover, within the domain of developer expertise, heuristic approaches that collect quantitative features, such as commit counts, API calls, and textual descriptions like comments and identifiers, have demonstrated their utility as potential indicators of developers' expertise. Yet, these approaches often lack a strong theoretical foundation and may be susceptible to biases.

CHAPTER 4 AN EMPIRICAL STUDY ON CODE GENERATED BY AI PAIR PROGRAMMING ASSISTANTS

4.1 Chapter Overview

The development of AI programming assistant tools incorporating LLM raises attention to evaluating their limitation and potential in automatically generating code to complete programming tasks. These tools are poised to function as AI pair programmers in software projects and the quality of the code they generate will inevitably influence the quality of software projects. The first step before proposing methodologies and techniques to enhance these tools is to comprehend their limitations. In this chapter, our focus centers on Copilot as a candidate due to its widespread adoption among developers as an AI pair programming tool [85, 101], and the possibility of using it as an in-IDE LLM-based coding assistant tool. We go beyond assessing the correctness of Copilot's suggestions and evaluate the quality of code generated by Copilot for certain programming tasks, comparing it to human-written code. We shed light on Copilot's limitations and capabilities and explore how, despite its limitations, it can effectively serve as a pair programming tool by leveraging the advantage of an SE tool to enhance the reliability of its suggestion. Also, we offer recommendations on how developers can leverage Copilot in practical scenarios. Lastly, we introduce a taxonomy of bugs occurring in code generated by LLMs and provide direction for future research in tailoring diverse SE quality assurance techniques to the characteristics of bugs generated by LLMs.

4.2 Study Design

In this section, we first define the objectives of this chapter. Following that, we detail both benchmarks that we use to evaluate GitHub Copilot and to assess the characteristics of bugs in LLM-generated code.

4.2.1 Setting Objectives of the Study

The objective of this study is first to assess the quality of the code snippets generated by Copilot beyond their correctness to highlight its limitations and potential. We aim to investigate the feasibility of employing Copilot as an effective AI pair programmer in software projects without compromising code quality and determine if Copilot's code can meet or surpass the quality of code generated by humans. We employ various quality metrics for dif-

ferent purposes and highlight the limitations of Copilot that may impact both the correctness and quality of its suggestions. Our focus is not on the type or difficulty level of programming tasks Copilot can handle, but rather on the quality of the code it would contribute to software projects as an AI pair programmer.

Furthermore, considering the prevalence of buggy suggestions from AI assistant tools like Copilot, we intend to explore the potential of using automated program repair techniques to enhance the reliability of Copilot's code suggestions. Our objective is to determine the extent to which these techniques can successfully repair buggy code generated by Copilot, thereby reducing the verification burden on developers and enhancing the overall reliability of Copilot's suggestions.

Finally, inspired by our evaluation results on the cost of repairing buggy code generated by Copilot compared to human-written code, we delve into the characteristics of bugs produced by various LLMs and present a taxonomy for bugs found in code generated by LLMs.

Specifically, this chapter seeks to answer the following research questions:

- **RQ1:** What are the potentials and limitations of the quality of code generated by Copilot in comparison to human-written code?
- **RQ2:** What are the characteristics of bugs present in code generated by LLMs?

4.2.2 Data Collection

In this section, we first present our dataset to assess Copilot, followed by an explanation of the details of students' submissions for programming tasks within this dataset. Subsequently, we explain the dataset used to extract the characteristics of bugs generated by LLMs for constructing the bug taxonomy.

Dataset to assess Copilot

We selected a dataset [1], which contains students' submissions for various programming assignments in a Python programming course. This dataset also serves as a benchmark for Python bug repairs [165]. While the programming tasks within this dataset may not encompass all the programming tasks undertaken by professional developers, it offers several advantages that make it well-suited for our study.

Firstly, it includes student submissions for each assignment, categorized as correct and buggy, providing a valuable basis for comparison with human-written code, which is one of the primary objectives of this chapter. The students can be considered novice developers. The bugs

detected in students' code can also occur in professional development settings, as demonstrated by tools like FindBugs [166]. Although FindBugs was initially designed to identify issues in student code, it has also successfully detected bugs in production software systems [167].

Furthermore, for every assignment in this dataset, a series of unit tests, generated by humans, are available. These tests are beneficial in assessing the correctness of code generated by Copilot. Additionally, the task descriptions within this dataset are written by humans, reducing the likelihood of issues related to memorization [168].

This dataset comprises a total of 2442 "Correct" and 1783 "Buggy" student submissions for five Python programming assignments within a Python course. Table 4.1 provides a description of each programming task. Each task in the dataset includes a human-written problem description, one or more reference solutions, varying numbers of student submissions categorized as "Correct" and "Buggy", and a set of distinct unit tests for each task. On average, there are 9 tests per problem, serving to assess the functional correctness of the solutions.

This dataset has been employed in various studies. Prior to our research, a study by Ahmed et al. [169] utilized this dataset to characterize the benefits of adaptive feedback for errors made by novice developers. Subsequently, after we conducted our study on evaluating Copilot with this dataset, several other studies have also employed it to evaluate LLMs for different purposes [170–174].

Dataset to assess the characteristics of bugs in LLM-generated code

We conducted this part of the study using a benchmark dataset named CoderEval [175]. This dataset consists of 230 Python functions from 43 open-source projects selected from GitHub, each function being used to generate 10 code samples for a given LLM. CoderEval was evaluated using three different LLMs: CodeGen [176], PanGu-Coder [177], and Codex [56]. Two of these LLMs, CodeGen and PanGu-Coder, are open source and have been used in previous studies that harness the power of LLMs for code generation tasks [120, 178–180]. The third one, Codex [56], is the model behind Copilot.

The samples in CoderEval exhibit different "runnable levels", indicating the extent of dependency a function requires for effective execution. Out of the 6,900 Python samples in CoderEval (2,300 per LLM), our sampling was restricted to code fragments falling into one of the following runnable levels: "self_runnable" (without external dependencies), "slib_runnable" (dependent only on Python's standard libraries such as "os" or "json"), or "plib_runnable"

Table 4.1 An overview of the dataset [1]

	Task	Description	Correct	Buggy
q1	**Sequential Search**	Takes in a value "x" and a sorted sequence "seq", and returns the position that "x" should go to such that the sequence remains sorted. Otherwise, return the length of the sequence.	768	575
q2	**Unique Dates Months**	Given a month and a list of possible birthdays, returns True if there is only one possible birthday with that month and unique day, and False otherwise. Implement 3 different functions: unique_day, unique_month, and contains_unique_day.	291	435
q3	**Duplicate Elimination**	Write a function remove_extras(lst) that takes in a list and returns a new list with all repeated occurrences of any element removed.	546	308
q4	**Sorting Tuples**	We represent a person using a tuple (gender, age). Given a list of people, write a function sort_age that sorts the people and returns a list in an order such that the older people are at the front of the list. You may assume that no two members of the list of people are of the same age.	419	357
q5	**Top_k Elements**	Write a function top_k that accepts a list of integers as the input and returns the greatest k number of values as a list, with its elements sorted in descending order. You may use any sorting algorithm you wish, but you are not allowed to use sort and sorted.	418	108
Total			2442	1783

(dependent only on Python's standard and public libraries such as "numpy"). As our focus was on observing bugs generated by LLMs, we considered only those generated code fragments flagged as buggy in the dataset. This filtering resulted in 1,997 buggy code samples.

To streamline the manual labeling effort, we opted to sample our data. At a 95% confidence interval and an error rate of 5%, this necessitated the analysis of 323 samples. With three LLMs, we sampled 330 code fragments, ensuring an equal distribution of 110 fragments for each LLM.

4.3 Approach

In this section, we introduce our methodology for evaluating Copilot as an AI pair programmer and provide a detailed explanation of the experimental design that we employed to achieve our objectives. Figure 4.1 illustrates the different steps of our method followed to conduct this evaluation and enhancement. Lastly, we explain our method for analyzing bugs in code generated by LLMs and detail the process of extracting a bug taxonomy specific to LLM-generated code.

4.3.1 Solving Programming Problems with Copilot

As illustrated in Table 4.1, each assignment comes with a corresponding description provided to the students in the programming course. We use the same description for each programming task as a prompt. We adhere to Copilot's default configuration, which returns a maximum of 10 suggestions for a given task specification in each attempt. We accumulate the top 10 suggestions over five separate attempts, resulting in a total of 50 solutions for each task.

As detailed in the previous section, there exist various test cases for each task. To assess the functional correctness of Copilot's solutions, a solution is categorized as "Correct" if it successfully passes all the unit tests associated with its respective problem. Otherwise, it is categorized as "Buggy".

Downsampling Student Solutions

The dataset contains a substantial number of student submissions for each task, exceeding 50 submissions, an average of 689.8 student submissions per programming task. To ensure a fair comparison, one possible approach would be to generate more suggestions using Copilot. However, increasing the number of attempts beyond 5 would lead to a higher occurrence of

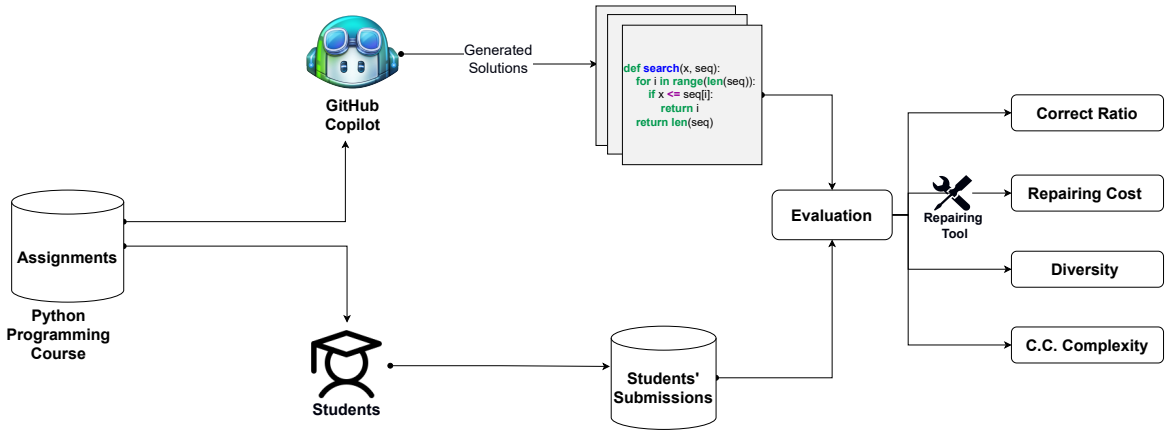


Figure 4.1 The workflow of our methodology

duplicate answers in Copilot's suggestions. We delve into the diversity of solutions in more detail in Subsection 4.3.2.

4.3.2 Evaluation Criteria

We evaluate solutions suggested by Copilot and human-written code for solving the programming tasks in the dataset based on various criteria. In the subsequent section, we provide a detailed explanation of each metric and the rationale for selecting it as an evaluation criterion.

1. Correct Ratio (pass@Topk)
2. Repairing Costs
3. Diversity
4. Cyclomatic Complexity

(1) Correct Ratio (CR@Topk)

A widely used metric for evaluating programming language models is the pass@k metric [56, 181]. This metric represents the fraction of correct solutions among the entire set of suggestions generated by LLMs, as indicated in equation 4.1, where n represents the total number of code snippets generated by LLMs, and "c" represents the number of solutions that passed all test cases when collecting the top k suggestions. However, due to Copilot's default configuration, which only returns the top 10 solutions in each attempt, we cannot accurately utilize this metric in our study.

$$pass@k = \mathop{\mathbb{E}}_{problem} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right] \tag{4.1}$$

In this study, our primary focus lies in the Correct Ratio (CR) obtained after collecting the top k suggestions in each attempt on Copilot. We refer to this metric as CR@TopK. Specifically, CR@TopK stands for the proportion of top k solutions collected from all attempts that successfully passed all the test cases when using Copilot with the same prompt on each attempt. For instance, when calculating CR@Top2, we collect the top 2 suggested solutions of Copilot for a problem in five different attempts ($\#solutions = k * n = 2 * 5 = 10$). CR@Top2 then reports the fraction of these solutions from whole solutions of all attempts that pass all test cases.

However, it's worth noting that CR@TopK can be calculated for Copilot, but it cannot be applied to students since there is no concept of "top k" in student submissions. For students, we calculate the fraction of correct submissions from their down-sampled submissions. In the case of Copilot, this metric closely resembles its CR@Top10.

(2) Repairing Costs

Measuring the cost associated with repairing bugs in a software project serves as a vital metric for assessing the quality of code snippets [182]. The long time required to repair a bug can often be linked to underlying structural issues within the code [182]. One significant factor impacting bug repair time is code churn, which refers to the size of changes needed to repair the bug [16]. Complex or low-quality code, especially when plagued with bugs, demands more time for repair. Also, developers may need extra time to detect the bug or larger patches may be necessary to resolve it.

Our observations indicate that Copilot sometimes generates buggy solutions that closely resemble correct solutions, requiring only a small modification to become correct. To gain perspective on the effort needed to repair buggy code generated by Copilot, we focus on assessing the overlap between its correct and buggy solutions, rather than functional correctness. As such, we calculate the BLEU score [183] (explained in chapter 2, between the AST of buggy code (provided it is compilable), and the AST of correct code generated by Copilot in a pairwise comparison. In cases where there is a syntax error in the buggy code and it cannot be converted to its AST, we calculate the BLEU score based on their lexical similarity instead.

Automatic Program Repair (APR) Tool: As an additional quality metric for comparing

code generated by Copilot with human-written code (students' submissions) and to gauge the extent to which the buggy code generated by Copilot can be repaired, we employ a SE Automatic Program Repair (APR) tool, *Refactory*, that we explained its details in Chapter 2. We then proceed to compare the repaired versions of the buggy solutions in terms of their repair costs. This assessment also explores the potential of existing SE tools to enhance the reliability of code generated by Copilot. We opted for an APR tool for the following reasons:

- Using an APR tool for repairing bugs is a common practice in software projects, leading to time savings and increased developer productivity [184].
- Employing an APR tool helps mitigate potential bias in the repair process that might arise from relying only on one specific human expertise.

Refactory provides the following metrics to assess the cost of repairing buggy solutions:

- **Repair Rate:** This metric indicates the fraction of buggy code that successfully passes all test cases after undergoing the repair process.
- **Avg. Repair Time:** This metric represents the average time required to repair a buggy program, measured in seconds.
- **Relative Patch Size (RPS):** This metric is defined as the Tree-Edit-Distance (TED) between the AST of a buggy code and the AST of its repaired code, normalized by the AST size of the buggy code.

(3) Diversity

While increasing the number of generated solutions by LLMs for a programming task results in an increased number of correct solutions passing all test cases [56, 181], it is not known that this leads to a broader array of solution variations or merely replicating previous ones.

Temperature is a parameter in the functioning of LLMs that controls the level of randomness or uncertainty in the model's output. When the temperature is high (e.g., close to 1), it makes the LLM more creative and generates a wider range of suggestions. This heightened randomness can lead to diverse and sometimes unpredictable responses. However, when the temperature is low (e.g., near 0), the LLM becomes more deterministic, providing more focused and less varied suggestions that align closely with the training data.

While GitHub Copilot claims to eliminate duplicate solutions from its top 10 suggestions, our observations reveal the presence of duplicated solutions within its top 10 suggestions in

a single attempt when the temperature is set to its default value of 0.8. Figure 4.2 illustrates three distinct solutions generated by Copilot for the task named "q3: Duplicate Elimination" during a single attempt. As demonstrated, all three code structures are identical. The only distinction between Figure 4.2a and Figure 4.2b lies in the variable names, namely "item" and "i". Furthermore, the solution showcased in Figure 4.2c mirrors the one presented in Figure 4.2a, albeit with additional comments. This discrepancy stems from Copilot's reliance on token sequence comparison to identify and eliminate duplicates.

To delve further into this inquiry, we undertake a comparison of the Abstract Syntax Trees (ASTs) of solutions, which includes different suggestions by Copilot and submissions from students, and then calculate the AST similarity between every two code snippets [185]. This entails the creation of the AST for each code snippet, followed by the removal of leaves linked to variable or function names. Additionally, we exclude comments, docstrings, or any natural language text, as these are extraneous to the code's structure. AST similarity is bounded between 0 to 1, where a score of 1 indicates structural equivalence between code snippet, regardless of their semantic similarity, while a score of 0 signifies no equivalence between programs. When the similarity score between two ASTs equals 1, they are deemed duplicates, and we retain only one of the solutions. Any value below 1 signifies a functional distinction between the two solutions.

To investigate the divesity in Copilot's suggestions, we assess whether considering all solutions within a single attempt and also increasing the number of attempts, thereby expanding the pool of solutions, enhances the number of unique suggestions generated by Copilot. Furthermore, we compare the diversity of solutions, both correct and buggy, provided by Copilot with those generated by students. This metric also evaluates Copilot's ability to introduce novel solutions in comparison to students when addressing a programming task.

(4) Cyclomatic Complexity

A programming language consists of a set of programming keywords, built-in functions, methods, and types. When solving a simple programming task, developers have the flexibility to choose from various programming keywords and built-ins to arrive at a solution. However, while this flexibility in completing programming tasks is desirable, it can impact code efficiency, readability, and even maintainability in certain cases [186,187]. In industrial projects, one of the objectives of pair programming is to transfer such expertise from expert developers to novice developers [188–190]. For example, Figure 4.3 presents two different solutions to the programming task "q4: Sorting Tuples" from Table 4.1. Code Sample #1 represents a submission from a student, while Code Sample #2 is generated by Copilot.

```python
def remove_extras(lst):
    new_lst = []
    for i in lst:
        if i not in new_lst:
            new_lst.append(i)
    return new_lst
```

(a) Sample code 1

```python
def remove_extras(lst):
    new_lst = []
    for item in lst:
        if item not in new_lst:
            new_lst.append(item)
    return new_lst
```

(b) Sample code 2

```python
def remove_extras(lst):
    # Duplicate elimination
    new_lst = []
    for item in lst:
        if item not in new_lst:
            new_lst.append(item)
    return new_lst
```

(c) Sample code 3

Figure 4.2 Different solutions generated by Copilot for the "q3: Duplicate Elimination" Task in one attempt.

To assess whether Copilot's suggestions are as readable as those generated by humans, we calculate the Cyclomatic Complexity CC (explained in Chapter 2) of Copilot's solutions and compare them to the CC of students' submissions for the same programming tasks. For instance, in Figure 4.3, the CC values for code samples #1 and #2 are 4.13 and 1, respectively. While code sample #1 involves two nested for-loops to sort the list, code sample #2 simply calls the "sort" function and employs the "lambda" expression to iterate over the list. This approach aligns more with Pythonic idioms and is generally more understandable.

This evaluation allows us to assess whether Copilot can provide code that is readable and maintainable (indicated by a lower CC) when used as a pair programmer in a software project. To perform these calculations, we use the RADON Python package [1]. It's worth noting that a CC value close to or exceeding 10 is typically considered indicative of code that doesn't adhere to best practices.

4.3.3 Introducing the characteristics of bugs in LLM-generated code

This section explains our methodology for assessing the characteristics of bugs in LLM-generated code. We conducted a manual analysis of all buggy samples employing an open coding procedure [191], similar to methodologies applied in prior studies [192, 193].

Open coding in the context of qualitative research and grounded theory involves extracting values for quantitative variables from qualitative data to enable some form of quantitative or statistical analysis [191].

Open coding, also known as postformed code, refers to the coding method by which codes or labels are generated during the coding process. This method proves to be more useful in software engineering studies when compared to preformed code, where a predefined list of codes is established before the coding process, based on the study's purpose [192, 193]. When employing the open coding approach, codes are subject to dynamic adjustments—they can be added, removed, or merged to develop the final codebook. This flexibility allows for a

[1] https://radon.readthedocs.io/en/latest

Code Sample #1 for q4

```
def sort_age(lst):
a = []
for i in lst:
    if a ==[]:
        a.append(i)
        continue
    else:
        for k in range(0,len(lst)):
            if i[1]>a[0][1]:
                a.insert(0,i)
                break
            elif a[-1][1]>i[1]:
                a.insert(len(lst),i)
                break
            elif i[1] > a[k+1][1] and i[1]<a[k][1]:
                a.insert(k+1,i)
                break
return a
```

Code Sample #2 for q4

```
def sort_age(lst):
lst.sort(key=lambda x: x[1], reverse=True)
return lst
```

Figure 4.3 Two different solutions to solve "q4: Sorting Tuples". Code Sample #1 has written by student and #2 is generated by Copilot.

more adaptive and nuanced analysis compared to the rigid structure of preformed codes [191].

In the open coding approach, two human reviewers independently start the process of coding on the same set of samples. During this phase, the reviewers explore patterns, similarities, and categories within the samples relevant to the quantitative variables [191]—characteristics or root causes of bugs in our study. They assign discrete categories to each sample. The two reviewers then engage in a discussion session to exchange and deliberate on their identified categories, forming the initial codebook.

In each discussion, a third reviewer, not involved in the coding process, intervenes in cases of conflict between the two reviewers to finalize the labels. The two reviewers continue to independently label the remaining samples and address any conflicts that arise during subsequent discussion sessions [193]. Through this iterative process, the two reviewers independently label all the samples. At the conclusion of the method, they report the agreement reached through these independent labeling processes before entering into any negotiations [193].

Following the same procedure, we identified various types of bugs based on our observations from the buggy samples. The aim was to inductively construct bug categories in a bottom-up manner through manual bug analysis.

The manual construction of the taxonomy involved three human reviewers, two of whom are senior Ph.D. students with experience in SE research and several years of industrial experience as software engineers. The third reviewer is a senior researcher with 10 years of research experience in the field of SE and AI.

We developed the codebook of taxonomy in two phases. In the first phase, we selected 10% of the sample set, and two reviewers independently labeled buggy samples by their own defined descriptive labels. Subsequently, the entire team convened to discuss the categories, resolve conflicts, and merge the individual categories to create a consolidated taxonomy, which served as the initial codebook for the next phase. In the second phase, we divided the remaining buggy code in the sample set into five parts, each containing 20% of the remaining buggy codes. Two reviewers manually investigated all samples in each part, and then the entire team met to cross-check the labels and address conflicts while resolving discrepancies. Whenever a new category emerged (i.e., a bug could not be classified under the existing categories), the entire team gathered to discuss the new category, incorporate it into the codebook taxonomy, and re-label the affected samples. We observed several buggy samples that contained multiple types of bugs. In such cases, we classified the buggy sample in all identified bug types. The entire process required approximately 108 person-hours and resulted in a taxonomy comprising 10 bug categories. To assess inter-rater agreement, we used Cohen's Kappa metric resulting in the agreement of 68.16% for independent labeling between two main reviewers. Any conflicts were thoroughly discussed until a 100% agreement was reached [193].

4.4 Study Results

In this section, we discuss our findings to answer each research question of this chapter separately.

RQ1: What are the potentials and limitations of the quality of code generated by Copilot in comparison to human-written code?

To answer this research question, we evaluate the quality of code generated by Copilot and compare it with human-written code. We will explore the results for each evaluation criterion separately.

(1) Correct Ratio (CR@Topk)

As detailed in Subsection 4.3.2, we calculate the CR@Topk for solutions generated by Copilot for each programming task and also across different programming tasks. The CR@Topk metric represents the proportion of correct solutions among the Topk solutions, which are gathered across 5 different attempts.

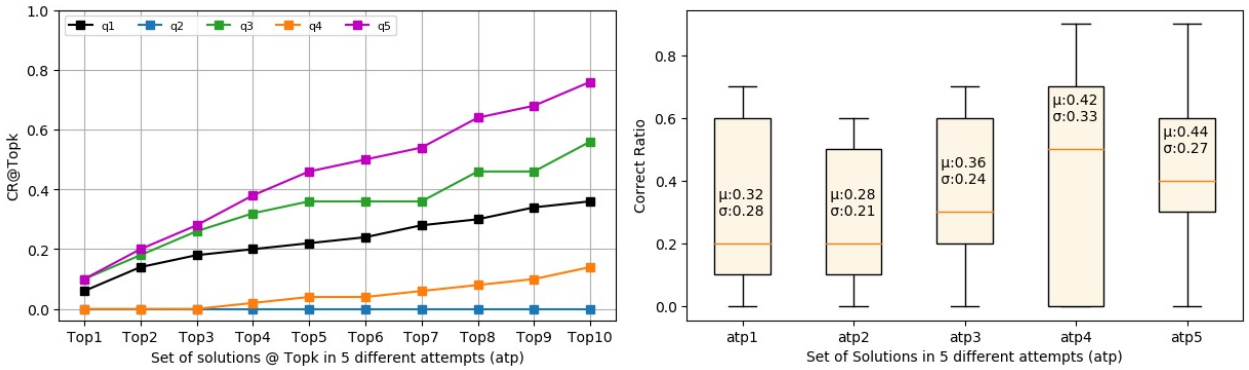
Figure 4.4a presents the CR@Topk results of Copilot's suggestions for each programming task across different attempts. The Topk solutions range from Top1 to Top10 since each Copilot attempt, in its default configuration, provides only the top 10 suggestions. For "q4: Sorting Tuples", Copilot did not yield any correct solutions in its top three suggestions across 5 different attempts. However, the CR@Top4 improves to 0.02 for this task. For "q3" the CR remains constant at 0.36 when collecting solutions from Top5 up to Top7. In contrast, "q5" exhibits the highest CR@Topk values, particularly for Top4 and beyond. "q4" has the lowest CR@Topk, following "q2".

Furthermore, Figure 4.4b illustrates the CR of solutions in each attempt independently. While the distribution of CRs varies across different attempts, attempting multiple times can increase the average CR of solutions. For example, the average CR in the first attempt (atp1) is 0.32, but this increases to 0.44 in the last attempt (atp5).

Table 4.2 provides a comparison of CR@Top1, CR@Top5, and CR@Top10 for solutions generated by Copilot with the CR of students' submissions. The CR of students' submissions is derived as the average of 5 down-sampling, and it consistently exceeds the CR of Copilot across various TopK values for all programming tasks with an average of 0.59. These results also highlight that increasing the number of samples from Copilot enhances the likelihood of encountering more correct solutions.

It's worth noting that Copilot failed to generate any correct solutions for the "q2: Unique Dates Months" programming task. This task explicitly requires the implementation of three distinct functions, as stated in the task description: *"...solve the problem by implementing 3 different functions..."*. However, Copilot produced code that addressed the task within a single function. Consequently, all of Copilot's solutions for this task failed the test cases, as the test cases for this task are designed to evaluate the implementation of three separate functions.

Another example can be observed in the "q4: Sorting Tuples" task. The task description specifies: *"...sort the people in an order such that the older people are at the front of the list..."*. However, Copilot failed to address this specific detail in the prompt and retained the default sorting order, which is in ascending order, in the majority of its generated code. As a result,



(a) CR@Topk of 5 different attempts

(b) CR of solutions in 5 attempts

Figure 4.4 Evaluation of correct solutions generated by Copilot.

the CR@Top10 for this task is only 0.14. In contrast, in the "q5: Top-K Elements" task, where the prompt instructs: *"...its elements sorted in descending order..."*, Copilot correctly generated code that adhered to the specified sorting order, resulting in a CR@Top10 of 0.76. However, it's worth mentioning that while Copilot considered the correct sorting order in this example, it overlooked another critical aspect of the prompt, which could not be detected by test cases. The task specification explicitly stated: *"...you are not allowed to use sort and sorted..."*, yet Copilot failed to adhere to this requirement, as all of its correct solutions utilized these two Python built-in functions.

In general, CR@Topk exhibits an upward trend as the number of samples (k) increases. This implies that collecting a larger number of solutions suggested by Copilot results in a greater number of correct solutions. It's important to note that this growth can differ based on various factors, including the nature and difficulty of the programming tasks, as well as the specific details provided in the prompt. In the remainder of this section, we delve further into the diversity of these correct suggestions.

(2) Repairing Costs

In this section, we evaluate the repair cost of buggy solutions generated by Copilot and compare it with students' buggy submissions. To measure the overlap between correct and buggy solutions generated by Copilot, we calculate the BLEU score based on the AST of the buggy and correct code. For buggy code snippets that contain syntax errors, we compute the BLEU score based on their lexical similarity.

Figure 4.5 illustrates the density distribution of BLEU scores among pairs of buggy and correct solutions generated by Copilot for different programming tasks. As observed in the

Table 4.2 The CR of Copilot's solutions and students' submissions

		Copilot			Students
	Task	**CR@Top1**	**CR@Top5**	**CR@Top10**	**CR**
q1	**Sequential Search**	0.06	0.22	0.36	**0.57**
q2	**Unique Dates Months**	0.00	0.00	0.00	**0.40**
q3	**Duplicate Elimination**	0.1	0.36	0.56	**0.64**
q4	**Sorting Tuples**	0.00	0.04	0.14	**0.54**
q5	**Top-k Elements**	0.1	0.46	0.76	**0.79**
	Total	0.05	0.22	0.36	**0.59**

figure, there are samples across different programming tasks where pairs of correct and buggy solutions exhibit BLEU scores of 0.75 or higher. This suggests that in some cases, making a minor alteration to a buggy solution generated by Copilot can readily transform it into a correct solution. For example, a simple change such as replacing "$>$" (greater) with "$\geq$" (greater than or equal to) can make the difference.

To evaluate the extent to which the buggy code snippets generated by Copilot can be repaired and to compare the associated repair costs with those of students, we used an APR tool, as discussed in Section 4.3.2. The results, as shown in Table 4.3, reveal varying repair rates and costs for Copilot and students across different tasks.

For instance, in the "q1: Sequential Search" task, the APR tool successfully repaired 94% of the buggy code generated by Copilot, compared to a 98% repair rate for students. The repair process for Copilot took an average of 9.61 seconds, while for students, it was notably faster at 2.58 seconds. The average repair patch size (RPS) was very close, with Copilot at 0.48 and students at 0.40.

In contrast, for "q2: Unique Dates Months", where Copilot generated no correct solutions (CR@Top10 = 0), the APR tool managed to repair 92% of the buggy code. In "q4: Sorting Tuples", where Copilot failed to consider the correct sorting order specified in the prompt, the APR tool repaired 100% of the buggy code generated by Copilot, achieving a swift average repair time of 0.78 seconds and an average RPS of 0.15.

Similarly, in "q5: Top-k elements", the APR tool repaired 100% of the buggy code generated by Copilot. However, the average RPS for Copilot's buggy code in this task was higher than that of students (0.50 compared to 0.30). It's important to note that this repair process focused solely on functional correctness and did not address adherence to the prompt's specification, such as the restriction on using "sort" and "sorted".

On average, the repair rate for Copilot's buggy solutions exceeded that of students, with rates of 95% and 89%, respectively. This suggests that, on average, 95% of buggy solutions

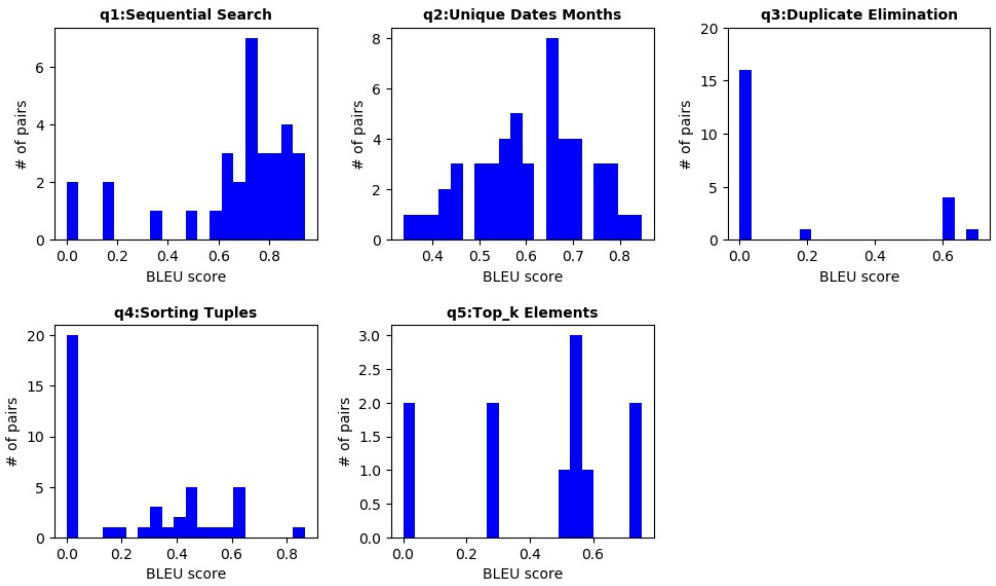


Figure 4.5 Distribution of BLEU score among the pair of correct and buggy solutions generated by Copilot.

generated by Copilot were successfully repaired. Moreover, while the average repair time and RPS varied when comparing Copilot and students across different tasks, the trend showed that repairing Copilot's buggy code was generally faster (in seconds) and resulted in smaller patches (smaller RPS). This underscores the advantage of using an APR tool to enhance the reliability of code generated by Copilot, particularly for bugs that are relatively easy to fix.

In conclusion, the high BLEU scores between correct and buggy solutions generated by Copilot indicate that many buggy code instances can be transformed into correct code with minimal effort. This effort is further reflected in the average repair time and average RPS reported by the APR tool as well. Although, on average, the CR of students' submissions surpasses that of Copilot's solutions, implementing an APR tool can repair up to 95% of buggy code generated by Copilot, ultimately elevating its CR. Therefore, the utilization of an APR tool can significantly improve the reliability of suggestions from AI assistant tools like Copilot.

(3) Diversity

While Copilot claims to eliminate duplicate solutions from its top ten suggestions, our results depicted in Figure 4.6 reveal the continued presence of duplicates within these suggestions. This Figure illustrates the cumulative distribution of Correct (C) solutions, None Duplicate Correct (NDC) solutions, Buggy (B) solutions, and None Duplicate Buggy (NDB) solutions

Table 4.3 Comparing the Repairing Cost of Copilot's suggestions with students's submissions

Task		Copilot			Students		
		Rep Rate	Avg Rep Time(sec)	Avg RPS	Rep Rate	Avg Rep Time	Avg RPS
q1	**sequential Search**	0.94	9.61	0.48	0.98	2.58	0.40
q2	**Unique Dates Months**	0.92	3.26	0.28	0.82	3.81	0.44
q3	**Duplicate Elimination**	0.91	0.64	0.26	0.96	4.35	0.30
q4	**Sorting Tuples**	**1.00**	**0.78**	**0.15**	0.85	8.82	0.29
q5	**Top-k Elements**	**1.00**	10.40	0.50	0.85	12.84	0.30
	Total	**0.95**	**4.94**	**0.33**	0.89	6.48	0.35

generated by Copilot and students across different tasks. For instance, up to the third attempt on "q3: Duplicate Elimination", Copilot suggests 17 Correct (C) solutions, but only 2 of these are NDC solutions. This indicates that, even after generating more solutions and conducting more attempts, Copilot repeats these 2 correct solutions multiple times. Conversely, out of 14 correct solutions generated by students in the third attempt (atp3), 13 are NDC. This pattern also holds for buggy solutions.

Increasing the number of attempts on Copilot results in an increase in the number of NDC for "q1" and "q5", going from 2 to 17 and 7 to 14, respectively. However, for "q3" and "q4", this growth is more modest, from 1 to 2 and from 2 to 4, respectively. The number of NDC solutions from Copilot is either less than or equal to the number of correct solutions in each attempt for each task. This trend is also observed for Buggy Solutions.

The difference between C and NDC in student submissions is less pronounced than in Copilot's results. For instance, in "q3", the cumulative number of correct solutions generated by Copilot across different attempts exceeds that of students' submissions from various sample sets. However, the opposite is true for NDC solutions. In "atp5" the cumulative number of correct solutions generated by Copilot totals 28, while it is 22 after 5 sample sets for students' submissions. Nonetheless, the cumulative NDC solutions for these attempts amount to only 2 (out of 28) for Copilot, while it reaches 21 (out of 22) for students.

As an additional example concerning Copilot, there are no more NDC solutions after "atp3" for "q3" and "q5." This implies that, as we increase the number of solutions generated by Copilot for these two questions, the CR increases primarily due to the duplication of correct solutions rather than the generation of new ones.

In general, the diversity of correct and buggy submissions from students surpasses that of Copilot. While there is no guarantee that all non-duplicate solutions are optimized, students have solved these 5 tasks with a greater variety of solutions.

In conclusion, despite Copilot's claims to remove duplicate solutions of each attempt, there

are evidently duplicates within the Top 10 solutions of each attempt. As discussed in Section 4.3.2, this discrepancy may arise from Copilot's reliance on token sequence comparison to identify and remove duplicates. Therefore, as we increase the number of sampled codes, there is indeed an increase in the fraction of correct solutions, as reported in different studies. However, this increment is significantly influenced by the generation of duplicate correct solutions, which may differ in token sequence due to varying variable names or additional comments but ultimately follow the same structure. Our results demonstrate the occurrence of duplicate solutions both within a single attempt and across multiple attempts with Copilot-generated code.

(4) Cyclomatic Complexity

In this section, we calculate the Cyclomatic Complexity (CC) of code generated by Copilot and students. Table 4.4 displays the average and standard deviation of CC for the correct solutions generated by Copilot and students.

On average, the correct solutions suggested by Copilot are found to be less complex than those provided by students. However, it's worth noting that, for example, Copilot has no correct solutions for "q2: Sequential Search" or the CR of Copilot for "q4: Sorting Tuples" is only 0.14% which can impact the average of CC.

Moreover, in "q5: Top-K Elements", Copilot used Python built-in functions "Sort" and "Sorted", despite the task's description explicitly instructed not to use them. The low CC observed in our results for both Copilot and students can primarily be attributed to the moderate difficulty level of the tasks in our dataset. However, the minor variances observed can be attributed to suboptimal coding practices evident in the students' solutions, as an example demonstrated in Figure 4.3.

Table 4.4 The CC of Copilot's solutions compare to students' submissions

	Question	**CC Copilot**	**CC Students**
q1	Sequential Search	5.8 ± 1.94	4.63 ± 2.1
q2	unique dates Months	-	4.18 ± 1.03
q3	Duplicate Elimination	3 ± 0.01	3.12 ± 0.5
q4	Sorting Tuples	1 ± 0	4.13 ± 1.03
q5	Top_k Elements	1.44 ± 0.69	3.3 ± 1.46
	Total	**2.81**	3.87

In the context of using Copilot as an AI pair programmer, it's worth noting that although the diversity of its correct solutions may not align with what we observed in student solutions,

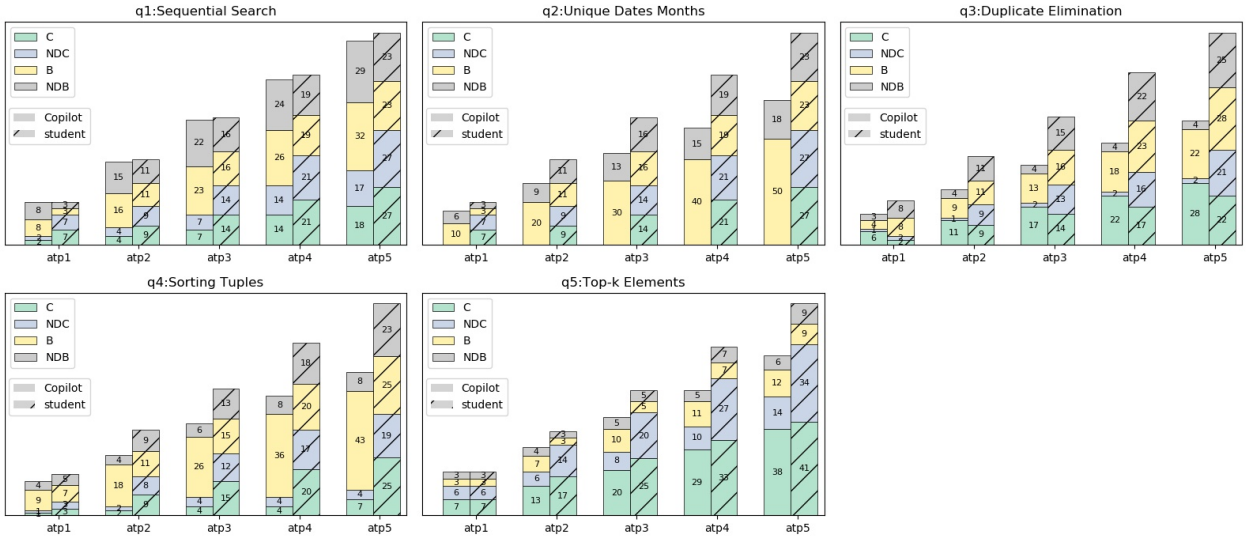


Figure 4.6 The cumulative distribution of solutions by Copilot and students before and after removing duplicates in four categories of Correct (C), Non-duplicate Correct (NDC), Buggy (B), and Non-duplicate Buggy (NDB).

as discussed in the previous section, its solutions stand out for their readability, ease of comprehension, and ease of modification due to their low CC.

RQ2: What are the characteristics of bugs present in code generated by LLMs?

The results of our previous research question indicate that Copilot generates more buggy code than novice humans. However, the cost of repairing its buggy code differs from that of human-generated ones. To identify and characterize the nature of bugs in code generated by LLMs, we conducted a manual analysis by examining the characteristics of these buggy code instances, as explained in Section 4.3.3.

In this section, we delve into the taxonomy of bugs observed within our benchmark dataset. We explore the nature, frequency, and distribution of these bugs both overall and within three different LLMs. Table 4.5 presents the characteristics of 10 bugs extracted from this study along with their descriptions. In Appendix A, we provide illustrative examples for each category in this taxonomy. These bug categories are defined based on the root cause of the bugs. As we mentioned in our methodology, some of the buggy code samples are labeled in multiple categories due to the presence of different types of bugs. On average, 62.76% of the samples are labeled in a single category, 29.12% with two, 8.4% with three categories, and only one instance of buggy code with four labels.

Table 4.6 displays the distribution of buggy code across different categories and by various

Bug Type	Description
Misinterpretation	The generated code does not align with the intended task prompt (when the model encounters difficulty in comprehending the task specifications).
Syntax Error	Any syntax error, such as a missing parenthesis, etc.
Silly Mistake	Mistakes such as redundant conditions or unnecessary casting. The issues will not introduce a bug on their own, but they can easily lead to one.
Prompt-biased code	LLM excessively rely on provided examples or words in the prompt while implementing a function, hindering the generalization of the generated code.
Missing Corner Case	The generated code functions correctly except for neglecting some corner cases.
Wrong Input Type	LLM uses an incorrect input type in a function call.
Wrong Function Call	LLM hallucinates by attempting to use a function that does not exist and has not been defined.
Wrong Attribute	LLM uses an incorrect attribute for an object or module or hallucinates by attempting to use an attribute that does not exist.
Incomplete Generation	LLM generates no code but instead produces, for example, a "pass" statement or an empty function.
Non-prompted Consideration (NPC)	LLM adds statements to the code that are unrelated to the task specification, resulting in errors or serving as the root cause of bugs.

Table 4.5 Bug Taxonomy for LLM-Generated Code

LLMs. Some categories are more prevalent in specific models. For instance, in the case of a robust model like Codex, the most common type of bug is “Missing Corner Case”, with 23.53% of its buggy samples. “Missing Corner Case” occurs when the model generates code that addresses the task description but fails on one or a few exceptional test inputs due to a minor oversight, such as using greater “>” instead of greater equal “>=”. Appendix A-*Missing Corner Case* presents an example of this type.

Conversely, in the case of a relatively weaker and open source model like PanGu-Coder, the most common category is “Misinterpretation”. This type of bug occurs when the code generated by the model does not align with the prompt and deviates from the intended purpose of the task.

On the other hand, CodeGen has the highest number of buggy samples in the “Silly Mistake” and “Syntax Error” categories. Codex, as a strong model, exhibits the highest number of samples in the “Non-Prompted Consideration” category. This category pertains to situations

Type of Bug	CodeGen	PanGu-Coder	Codex	Total
Wrong Function Call	5.10%	15.11%	10.46%	9.63%
Incomplete Generation	13.78%	3.60%	9.80%	9.63%
Misinterpretation	22.96%	23.74%	15.03%	20.70%
NPC	6.12%	5.76%	11.76%	7.79%
Silly Mistake	14.80%	4.32%	7.84%	9.63%
Wrong Input Type	6.63%	8.63%	2.61%	5.94%
Missing Corner Case	5.61%	20.14%	23.53%	15.37%
Wrong Attribute	7.65%	10.07%	8.50%	8.61%
Synthax Error	9.18%	5.04%	3.27%	6.15%
Prompt-biased code	8.16%	3.60%	7.19%	6.56%

Table 4.6 The distribution of Bug types for three different LLM-Generated Code

where the model considers additional conditions or statements that were not requested in the prompt, leading to bugs. For example, the model might convert the final output of a function into an integer, which was not required in the prompt, or check if the output is a string and then apply further functionality. An example of this type can be found in Appendix A-*Non-Prompted Consideration*.

The second most common type of bug in code generated by PanGu-Coder is "Function Hallucination". This occurs when the model erroneously hallucinates the existence of a function that does not exist. An example of this type is provided in Appendix A, "Function Hallucination". Our observations reveal that when the model struggles to address the prompt, it sometimes resorts to calling an auxiliary function that is not defined, assuming that this function accomplishes the primary objective of the task. Usually, the function name is biased on the words used in the prompt.

In general, "Misinterpretation" is the most common type across all models. The second most common types are "Missing Corner Cases" and "Function Hallucination", with "Incomplete Generation" and "Silly Mistake" following closely in the third position. It is noteworthy that these types of bugs are rarely observed in code written by humans. For example, humans typically do not frequently call undefined functions ("Function Hallucination") or access attributes on objects that do not exist ("Wrong Attribute"). Some of these bug types can be detected by Python Linters embedded in various IDEs, such as VS Code, which further reduces the likelihood of encountering them in human-written code. For instance, in the "Silly Mistake" category, a common mistake is when the model generates an if/else condition but uses the same statement in both branches, which is not a common mistake in human-written code. This example is presented in Appendix A-*Silly Mistake*.

In conclusion, our findings introduce an initial taxonomy of bugs generated by LLMs in code generation tasks. These results encompass 10 bug categories commonly observed in code generated by three different LLMs. These identified bug types can serve as a valuable resource for future research focused on the evaluation of LLM-generated code and the improvement of programming assistant tools incorporating LLMs.

4.5 Discussion and Practical Suggestions

The findings of this chapter reveal several key observations and implications, shedding light on the strengths and limitations of Copilot as an AI programming assistant. Also, we introduce a taxonomy of bugs in code generated by LLMs.

One of our findings is that Copilot, on average, achieves a CR of 0.36 on generating code in addressing certain programming tasks, which is notably lower than the 0.59 CR observed in human-generated code. One limitation highlighted in our results is that Copilot may overlook subtle details within the prompts, leading to a significantly lower CR. For instance, in a specific programming task where the objective is specified as *"...sort the people in an order such that the older people are at the front of the list..."*, Copilot failed to generate code with the correct sorting order and it retained the default sorting order, which is ascending order, in the majority of its generated code. Consequently, many of Copilot's solutions for this task fail to pass the test cases resulting at very low CR, 0.14. In our proposed bug taxonomy, we categorized this characteristic as "Missing Corner Case".

In contrast, when another task explicitly calls for sorting in *descending* order, Copilot performs better, 0.76 CR, highlighting its capability to understand and implement such programming-specific instructions. It is important to note that Copilot is fine-tuned primarily for code generation. As a result, it excels when tasks are described using programming constructions, such as specifying *ascending* or *descending* sorting orders.

Another noteworthy aspect highlighted in our results is Copilot's tendency to miss specific task details that may not be detected by the test cases. For instance, when the task explicitly states, *"...you are not allowed to use sort and sorted..."*, Copilot consistently includes these restricted functions in its suggestions, despite achieving a relatively high CR of around 0.76. In our proposed bug taxonomy, we categorized this characteristic as "Silly Mistake". These results underline a challenge in the reliance on test cases alone for evaluating the correctness of code generated by Copilot. While functional correctness is crucial and can be evaluated by test cases, there are nuances and missing concepts that only humans can identify. Therefore, the involvement of expert developers is essential to assess whether the code fully aligns with

the intended behaviors and requirements.

Our results also highlight the limitation of Copilot when it is asked to generate three distinct functions within a single prompt. The code generated by Copilot failed to meet this requirement and instead implemented the entire prompt within a single function. Given that there were test cases designed to evaluate each of the three functions separately, the CR for this particular task turned out to be zero. In our proposed bug taxonomy, we categorized this characteristic as "Misinterpretation".

In an empirical experiment, we divided the initial prompt into three separate prompts, each focused on a single functionality. When we ran these individual prompts separately in Copilot, the CR increased significantly, reaching an average of 0.88. This practical exercise demonstrated that, as a best practice, it is more effective to present each required functionality as a separate prompt to LLMs.

Despite the relatively low CR, our results reveal a high degree of lexical similarity among various pairs of correct and buggy code generated by Copilot. These buggy solutions often fail to pass all test cases due to minor syntax errors (as "Syntax Error" in bug taxonomy) or the omission of specific corner cases from the task specifications (as "Missing Corner Case" in bug taxonomy). This observation, coupled with average repair times and patch sizes, highlights that minor adjustments can often transform buggy code generated by Copilot into correct solutions. By applying an APR tool to automatically repair the buggy code, the reliability of Copilot's suggestions can increase significantly, with a repairing rate of 95%.

As a supplementary result, our analysis of the characteristics of bugs occurring in code generated by LLMs provides insights into the nature, frequency, and distribution of these bugs by proposing a bug taxonomy for buggy code generated by LLMs. These analyses lead to interesting observations on different bug characteristics found in LLM-generated code, such as "Function Hallucination" and "Prompt-biased Code", which are not common in human buggy code. Our results serve as an initial taxonomy of bugs observed in LLMs' generated code. Further investigation is required to compare their frequency in code written by humans. These findings represent the first steps in characterizing bugs in LLM-generated code and open up avenues for future studies on enhancing the quality of code generated by programming assistant tools incorporating LLMs and also improving different SE tools that rely on characteristics of human buggy code, such as mutant operators in Mutation Testing (MT) tools or repair rules in APR tools.

Additionally, as highlighted by previous studies, increasing the number of solutions generated by LLMs for a programming task can indeed yield a higher count of correct solutions that pass all test cases [56, 181], our findings indicate that this increment does not necessarily

result in a wider array of solution variations. Instead, it often involves the duplication of previously generated solutions. However, Copilot claims to remove duplicate solutions in its top-k suggestions. This issue was also raised by different users on a GitHub discussion titled *"Copilot doesn't remove duplicate, while it says it does"*[2], where users shared instances of duplicate code samples that only differed in minor details, such as quotation marks. This discussion underscores the significance of our findings because the presence of such duplicates can have an impact on developers' satisfaction which is a crucial metric when assessing developer productivity, as per the SPACE framework [94].

In the context of Cyclomatic Complexity (CC), Copilot demonstrates a tendency to produce solutions with lower complexity in comparison to those generated by novice programmers (students). Furthermore, we observed proficiency in Copilot's usage of advanced programming keywords and its adherence to more Pythonic idioms when addressing programming tasks, as shown in Figure 4.3. This alignment with one of the primary goals of pair programming, which is to facilitate the generation of code that is both less complex and more maintainable, underscores Copilot's potential as a valuable aid in achieving this objective.

In conclusion, our findings reveal that programming assistant tools incorporating LLMs generate code that continues to exhibit issues, being buggy, duplicated, inefficient, containing syntax errors, or failing to align with the intended functionality. Nonetheless, the implementation of post-processing steps, such as repairing its buggy code, can significantly improve the reliability of its suggestions. However, it is important to emphasize that the validation of its final suggestions still requires the expertise of an expert developer. Furthermore, Copilot may pose challenges in SE projects if employed by novice developers, who may struggle to discern between its buggy suggestions, such as "Silly Mistakes" or "Missing Corner Cases", due to their lack of expertise.

4.6 Threats to Validity

We will now discuss the threats to the validity of our study following the guidelines provided by Wohlin [194] for experimentation in software engineering.

Internal Validity

One potential threat to internal validity arises from the fact that Copilot is a closed-source tool. We are unable to perform a detailed analysis of our results based on the specific characteristics and expected behavior of Copilot's trained model. Similarly, the lack of access

[2] `https://github.com/orgs/community/discussions/10683`

to Copilot's training data prevents us from determining whether it memorized solutions from its training set or generated unique solutions. In line with other researchers [17,56,85,88], our study focuses on investigating Copilot's functionality in suggesting code based on provided prompts.

Regarding the APR tool we employed [69], it requires at least one correct solution to facilitate the repair of buggy code. In contrast to some other APR tools that necessitate multiple correct solutions, the advantage of the APR tool we used lies in its ability to effectively operate with just a single correct solution. This approach is particularly valuable in scenarios such as student submissions, where there are either some correct code submissions from students or at least a single reference correct code for each programming task.

The APR tools designed for repairing assignments are also built on the assumption that a significant number of assignments contain errors, as opposed to APR tools used in real-world programming, which typically assume that codebases are mainly correct [70]. In our study, employing the APR tool for programming assignments is a good option since a considerable portion of the code generated by Copilot exhibited errors similar to those made by novice developers. Furthermore, in our dataset, we had access to one correct reference solution for each programming task, ensuring that the tool was not constrained by its limitations. However, future studies could explore alternative techniques in APR or even leverage LLMs to repair their own buggy code.

External Validity

One limitation related to external validity is the absence of a dataset from an industrial context that includes programming task statements along with their corresponding code. Consequently, we followed the path of previous research in software engineering by using classical programming tasks to assess Copilot's competence [17,18,87–89]. There are different advantages to these types of programming tasks that we discussed in Subsections 4.2.2. To highlight two advantages, first, Copilot is able to generate code corresponding to these task descriptions. Thus, we could apply our assessments beyond the correctness of the suggested solutions. Also, the task descriptions in our datasets are human-written and that mitigate memorization issues in LLM. But these programming tasks may not be representative of all programming tasks encountered in real software projects.

Considering the choice of programming tasks and to have a fair comparison, we compared Copilot with students in a Python programming course. While we have no information about the background and characteristics of the participants, we assume that they are good representatives of junior developers in real software projects, but they may not be representatives

of the whole population.

Conclusion Validity

To mitigate the conclusion validity of our study, we adopted various quantitative metrics, drawing inspiration from prior research in software engineering, to compare Copilot's code with that of humans [17, 79, 182]. While these quantitative metrics reduce the potential for biased conclusions, they do not allow for qualitative assessments, such as understanding how humans interact with the tool and their experiences when using Copilot as an AI pair programmer.

Construct Validity

A threat to the construct validity of our study lies in the fact that not all the features and capabilities of a proficient AI pair programmer can be effectively captured through quantitative metrics alone. Pair programming involves interactions between humans and between humans and tools, making it essential to consider human opinions regarding their experiences with such AI pair programming tools. For instance, some individuals may prefer a pair programming tool that accepts voice commands over one that provides a list of suggested solutions, as they value the discussion aspect of pair programming more than receiving solution suggestions. Evaluating user experiences and preferences can contribute to a more comprehensive understanding of AI pair programming tools.

4.7 Chapter Summary

In this chapter, we have delved into Copilot's capacity for code generation and conducted a comparative analysis between its generated code and those generated by humans to highlight its potential and limitations. Our findings reveal that Copilot's code has a lower CR compared to human code. This discrepancy arises from a combination of factors, including both the model's performance limitations and instances where Copilot overlooks certain prompt details or generates code that fails test cases due to minor syntax errors. Additionally, we employed an APR tool to repair the buggy code generated by Copilot. Our results indicate that the cost of repairing Copilot-generated code is lower than that of code produced by novice developers. The APR tool successfully repaired up to 95% of the buggy code generated by Copilot. These findings highlight the advantages of integrating post-processing steps into the code generated by tools like Copilot. Such steps enhance the reliability of Copilot's suggestions and reduce the time developers need to spend on verifying and rectifying potential issues. Lastly, we

introduce a bug taxonomy on the characteristics of bugs that we observed in LLM-generated code.

CHAPTER 5 ENHANCING THE EFFECTIVENESS OF TEST CASES GENERATED BY PRE-TRAINED LLMS

5.1 Chapter Overview

The previous chapter highlighted the capacity of LLMs as AI programming assistants to generate code and assessed the quality of their code in addressing programming tasks. Nevertheless, the results also revealed instances of low-quality code in the generated output, which may contain bugs or be suboptimal. Additionally, the findings have indicated that implementing post-processing steps, such as using an APR tool to repair buggy code, can enhance the reliability of code generated by such models. In this chapter, we pivot our focus to the automatic generation of test cases, a critical phase in SE that has long been a focal point of automation research. We introduce MuTAP (**Mu**tation **T**est case generation using **A**ugmented **P**rompt), a method primarily relying on an LLM core to generate test cases. MuTAP enhances the effectiveness and reliability of test cases generated by LLMs through a refinement process that leverages Mutation Testing (MT)—a well-known technique in SE for evaluating the effectiveness of tests in revealing bugs. Our objective is achieved by augmenting prompts with surviving mutants, as these mutants highlight the limitations of test cases in detecting bugs. Notably, MuTAP is capable of generating effective test cases even in the absence of natural language descriptions of the Program Under Test (PUTs). We employ different LLMs within MuTAP and assess their performance across different benchmarks. The effectiveness of test cases generated by MuTAP outperforms both the current state-of-the-art fully-automated test generation tool (Pynguin) and zero-shot/few-shot learning approaches on LLMs.

5.2 Study Design

In this section, we present an overview of the methodology employed by *MuTAP*, as illustrated in Figure 5.1. Our proposed technique, *MuTAP*, relies on an LLM Component (LLMC) as its primary core. The process begins by providing a prompt to the LLMC to generate test cases. The initial prompt comprises the *PUT* and instructions for generating test cases using both *zero-shot* and *few-shot* learning approaches.

Building upon the results from the previous chapter, which demonstrated that repairing bugs in code generated by LLMs can enhance their reliability, in this chapter after generating test cases with the LLMC, *MuTAP* conducts a syntax assessment of the generated test cases.

If any syntax errors are detected, *MuTAP* re-prompts its LLMC to fix syntax errors. To further enhance the reliability and effectiveness of the generated test cases, *MuTAP* repairs their intended behavior by replacing the unexpected output of the test oracles for specific test inputs to the expected return values of the *PUT* when subjected to the same test input.

Subsequently, *MuTAP* applies MT to assess the effectiveness of the test cases in identifying bugs within the *PUTs*. As surviving mutants highlight the limitation of the generated test cases, *MuTAP* re-prompts its LLMC to generate new test cases using those surviving mutants. This is achieved by augmenting the initial prompt with both the initial test cases and the surviving mutants. *MuTAP* halts the process of augmenting the initial prompt when either the final test cases can effectively detect all mutants or there are no surviving mutants left that have not already been used to augment the initial prompt.

We employ two types of LLMs as the LLMC of *MuTAP*: *Codex*, which is designed for code-related tasks, and *llama-2-chat*, which is optimized for dialog use cases and versatile enough to accommodate a range of tasks, including programming. We evaluate *MuTAP* on both synthetic bugs of 164 *PUT*s [56] and 1710 real buggy programs generated by humans collected from a Python bug repairing benchmark, the dataset used in the previous chapter [69]. Following this section, we outline the objectives of this chapter and provide a motivating example that elucidates the rationale behind this approach.

5.2.1 Setting Objectives of the Study

Testing is an important yet expensive step in the software development lifecycle. Generating effective tests is a time-consuming and tedious task for developers. The automatic generation of test cases is a significant area of interest in SE, aiming to reduce developers' testing efforts. Several studies have explored the potential of LLMs in test case generation [22, 113, 114, 195]. While the preliminary results of these studies are promising, they primarily focus on evaluating the effectiveness of test cases in terms of test coverage, without an emphasis on improving the bug detection capabilities of test cases. Furthermore, it is recognized in the literature that while test coverage is a useful metric for assessing test quality, it has a weak correlation with the efficiency of tests in bug detection [196–198].

In this chapter, our objective is to enhance the effectiveness of test cases generated by LLMs in terms of their bug-revealing capabilities through post-processing steps. We present the first study that leverages MT to improve and evaluate the effectiveness of test cases generated by LLMs for Python programs, specifically focusing on their ability to uncover faults rather than improving code coverage. Surviving mutants highlight the shortcomings of test cases and can serve as a guide for enhancing their bug detection capabilities. State-of-the-art

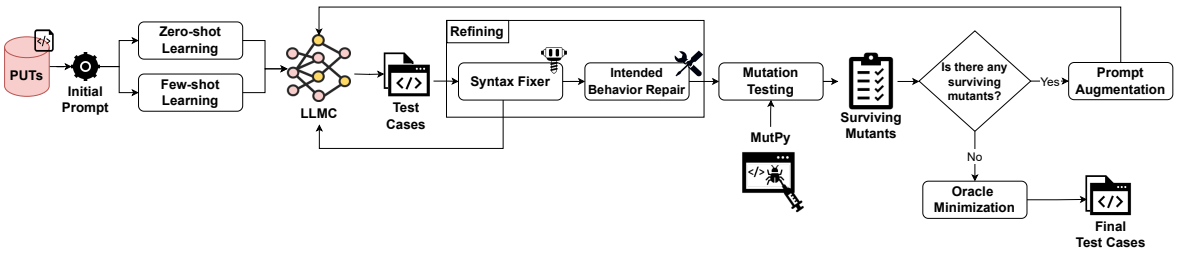


Figure 5.1 The proposed methodology for generating effective test cases using LLMs.

automatic test generation tools like Pynguin [111] cannot harness the potential hidden in these surviving mutants due to their limitations in code synthesis. We leverage the code synthesis capabilities of LLMs and utilize surviving mutants to improve the effectiveness of test cases in detecting bugs.

To summarize, this chapter makes the following contributions:

- We present the first study that generates test cases with LLMs by leveraging MT, a well-known SE technique for evaluating test effectiveness in bug detection.
- We introduce a prompt-based learning technique to enhance the effectiveness of test cases by augmenting prompts with both initial test cases and surviving mutants of a *PUT*. Our proposed technique is adaptable to new LLMs by replacing its LLMC without disrupting the entire process.

5.2.2 Motivating Example

In this section, we present an example in Figure 5.2 showing how our proposed approach generates effective test cases. Suppose we have 10 mutants $\{SM_0, SM_1, ..., SM_9\}$ for the *PUT* in Figure 5.2. The goal of our proposed technique, *MuTAP*, is to generate effective test cases for the example *PUT* in a way that ensures killing the maximum number of mutants.

The function *any_int()* in Figure 5.2 receives 3 inputs and returns *True* if all 3 inputs are integers, also one of the inputs is equal to the sum of two others. Otherwise, it returns *False*. In the first step, *MuTAP* uses the initial prompt, ①, to run a query on the LLM Component (LLMC) and generates initial test cases for this *PUT*. The component ② in Figure 5.2 shows the initial test cases generated by LLMC after the refining step. We named it Initial Unit Test, *IUT*. In Section 5.3.2, we discuss the refining step (syntax and intended behavior fixing) of our approach in detail. The *IUT* kills 6 out of 10 mutants of *PUT*. The 4 remaining mutants reveal the weaknesses of the generated test, meaning that *IUT* needs new test cases with assertion to kill the injected bugs in those 4 mutants.

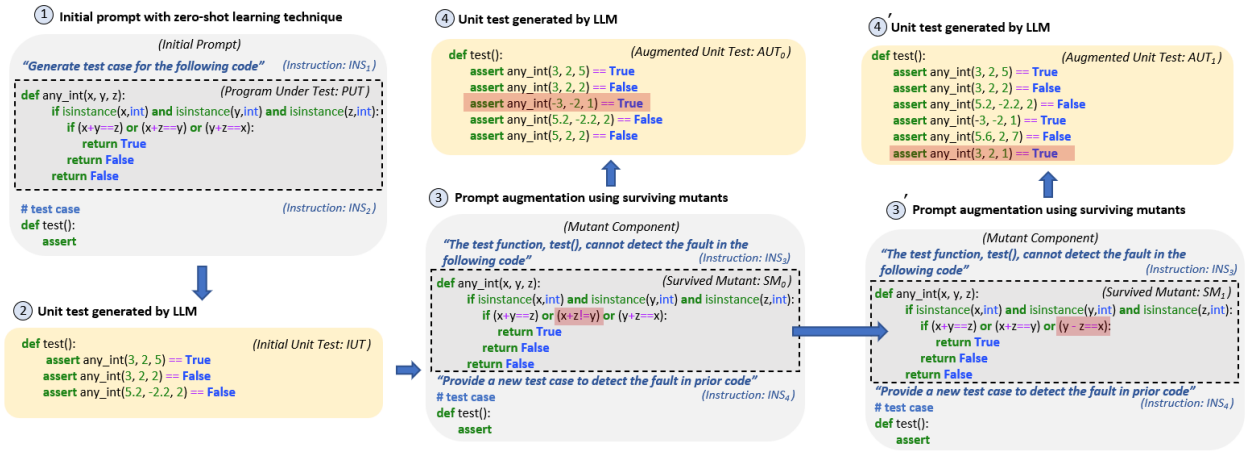


Figure 5.2 Different steps of *MuTAP* on a sample PUT.

To address this limitation and improve the effective test cases, *MuTAP* augments the initial prompt with two new components; the first one is the response of the model to the initial prompt after fixing its syntax and intended behavior, *IUT*, and the second one is the mutant component, ③ in Figure 5.2. *MuTAP* initiates the construction of the mutant component by using one of the "Survived Mutant" of *PUT* that we refer to as SM_0. The red highlight in SM_0 shows the injected bug in *PUT*. The injected bug changes the second statement in the condition of the inner *if* in *PUT* in a way that the sum of the first and last input of function *any_int()* is not equal to the middle input anymore. Since there is no test case in *IUT* to verify that its middle input, y, is equal to the sum of its first and last inputs, x and z, *IUT* is not able to kill this mutant.

MuTAP uses the concatenation of these three components: ①, ②, and ③ to re-prompt the LLMC. The ④ component in Figure 5.2, shows the new set of test cases generated by LLMC appended to *IUT* after the refining step. We named it Augmented Unit Test, AUT_0. The unit test has two more assertions compared to the *IUT* and one of them, highlighted in red, kills the mutant, SM_0.

MuTAP applied AUT_0 to the mutants of *PUT* again. If there are any remaining surviving mutants, *MuTAP* iterates the augmentation process by updating the mutant component with another surviving mutant if it has not been used to augment the prompt previously. *MuTAP* utilizes each mutant individually because sometimes new test cases that address one mutant can also kill the remaining surviving mutants. Moreover, due to the limited length of the prompt and non-constant length of mutants, applying each surviving mutant separately is a more practical approach. Figure 5.2 ③′ shows an example of how the mutant component is updated using another surviving mutant. We call this mutant SM_1. Unit test,

④′, shows a new set of test cases including one assertion that detects SM_1. *MuTAP* iterates the augmentation process until either the final test cases can kill all the mutants, or there are no surviving mutants left that have not already been used to augment the initial prompt.

The final test cases generated by our proposed technique, *MuTAP*, kill 9 out of 10 mutants of this example, *PUT*, and it increases the Mutation Score (MS) for *PUT* from 60% (6 out of 10) to 90% (9 out of 10). The MS quantifies the effectiveness of test cases by calculating the ratio of killed mutants to all mutants for a *PUT*. This result can be compared to the state-of-the-art automatic test generation tool for Python programming language [199], Pynguin, which generates a test case for *PUT* with only a 40% MS. This tool uses a search-based generation technique [200] and randomly mutates the test values within a test case to generate new test cases. The random nature of this method results in a low chance of generating a new test case that can kill the surviving mutants of *PUT*.

5.3 Approach

In this section, we will delve into the various steps of our approach for implementing *MuTAP*. Figure 5.1 provides an overview of our proposed methodology, while Algorithm 1 outlines the sequence of its distinct steps.

5.3.1 Initial Prompt

LLMs are capable of performing tasks for which they have been already trained. Fine-tuning LLMs for new tasks can be computationally expensive. Additionally, there are LLMs, like Codex, that demonstrate a very good performance in code generation. However, due to their closed-source nature, fine-tuning them for new tasks is not feasible.

Prompt-based learning [72, 201] is an effective technique for adapting LLMs to new tasks. A prompt is a combination of natural language and/or programming language context used as input for LLMs. Studies have shown that providing a natural language instruction as a hint (*zero-shot learning*) [22, 48, 114], or offering several examples (*few-shot learning*) [39, 60, 116] in the prompt enhances the capabilities of LLMs in performing new tasks.

MuTAP employs both *zero-shot* and *few-shot* learning to construct the initial prompt and applies LLMC to them separately. This step is illustrated in Algorithm 2. In more detail, our approach employs *zero-shot* and *few-shot* techniques as follows:

- *zero-shot*: The initial prompt generated using the *zero-shot* technique consists of three components, following the approach outlined in [22]. In Figure 5.2, component ①,

Algorithm 1: *MuTAP*

```
   Input: PUT, LLMC, initial_prompt_type
   /* INS_1, INS_2, INS_3, INS_4 and INS_fix are global variable as natural language instructions for the
      prompts                                                                                          */
   Output: FUT // Final Unit Test
   // Initial Prompt
 1 initial_prompt ← GenerateInitialPrompt (PUT, initial_prompt_type)
 2 raw_IUT ← LLMC (initial_prompt)
   // Syntax Fixer and Intended Behaviour Repair
 3 IUT ← Refining (raw_IUT, PUT)
   // Mutation Testing
 4 MS, surviving_mutant ← MutationTesting (PUT, IUT)
 5 if MS < 100% then
       // Prompt Augmentation
 6     AUT ← AugmentingPrompt (MS, PUT, initial_prompt, IUT, surviving_mutant)
       // Oracle Minimization
 7     FUT ← OracleMinimization (AUT)
 8 else
       // F: Oracle Minimization
 9     FUT ← OracleMinimization (IUT)
10 end
11 return FUT
```

provides an illustrative example of such a prompt. The first unit in this component is an instruction in the natural language referred to as INS_1, which clarifies the task by requesting: *"Generate test cases for the following code"*. The second unit is the Program Under Test (PUT), and the final unit comprises instructions in a programming language named INS_2. The purpose of INS_2 is to serve as a hint to specify the desired output for LLMC. The concatenation of (INS_1, PUT, INS_2) constructs the initial prompt for zero-shot learning (Line 2 in Algorithm 2).

- *few-shot*: Prompt generation using the *few-shot* learning approach involves a sequence of inputs and expected outputs related to the downstream task. Various methods exist for presenting these input-output pairs in the prompt. In our work, we adhere to the approach outlined in [60] to construct the initial prompt using the few-shot strategy within *MuTAP*. Given the maximum token length for LLMC (4k tokens in our study), the few-shot prompt includes two distinct demonstrative examples of a Method (M) and a Unit Test (UT) (Line 5 in Algorithm 2).:

  ```
  <code>M_1</code>\n<test>UT_1</test>\n <code>M_2</code>\n<test>UT_2</test>\n
  <code>PUT_i</code>\n <test>
  ```

In the initial prompt, step ① in Figure 5.2, there is no natural language description of the 'PUT' (Program Under Test). This omission is because such descriptions may not always be accessible, and *MuTAP* depends on the capability of LLMC to synthesize code context. *MuTAP* invokes the initial prompt, whether it's *zero-shot* or *few-shot*, on LLMC, and then, it passes the inferred output to the next step (Line 2 in Algorithm 1).

Algorithm 2: GenerateInitialPrompt

```
   Input: PUT, initial_prompt_type
   Output: initial_prompt
1  if initial_prompt_type == "zero-shot" then
2  |  initial_prompt ← CONCAT(INS_1, PUT, INS_2)
3  else
4  |  if initial_prompt_type == "few-shot" then
5  |  |  initial_prompt ← CONCAT(pair(M,UT), PUT) // M: Method, UT: Unit Test
6  |  end
7  end
8  return initial_prompt
```

5.3.2 Refining

In this section, we describe the process of refining the generated test cases within *MuTAP*, encompassing fixing syntactical errors and intended behavior repair. The specific steps are outlined in Algorithm 3.

Syntax Fixer

The test cases generated by LLMC may contain syntax errors, such as missing brackets or incomplete lines. As *MuTAP* requires the execution of the test function for further analysis in MT and prompt augmentation, samples with syntax errors can prove to be inefficient. However, based on our findings in Chapter4, sometimes a minor adjustment in LLMC's output can repair the syntactic errors, transforming them into executable code.

MuTAP harnesses the capabilities of its LLMC to address syntax errors, in line with similar studies [72, 202]. To accomplish this, *MuTAP* invokes LLMC with a new prompt designed to fix the syntax error present in its initial output (referred to as *SyntaxFixer* in Algorithm 3). The syntax-fixing prompt comprises two components. The first component is a natural language instruction, denoted as INS_{fix}, which instructs LLMC to *"Fix the syntax errors in the following code snippet"*. The second component consists of the test function generated by LLMC in response to the initial prompt (Lines 7-8 in Algorithm 3). If the syntax error persists even after re-prompting LLMC, *MuTAP* employs a Python parser to identify the erroneous line. It then retains the lines preceding the error, ensuring they remain free of syntax errors (Line 13 in Algorithm 3).

Intended Behavior Repair

Based on the initial prompt, LLMC generates various test cases, which are serialized with assertion oracles. These test cases are created by invoking the PUT with specific inputs and comparing the returned output to the expected output or ground truth, for example,

`{assert add(2,2) == 4}`. However, there is a possibility that LLMC may generate test cases with assertions that validate incorrect return values. This implies that for certain test cases, LLMC does not produce the expected return output for the *PUT* function. The absence of a natural language description of the *PUT* in the initial prompt could potentially lead to the generation of test cases that do not accurately capture the intended behavior of the method.

Assertions with incorrect return values may fail not because they detect a bug but due to unintended behavior within the assertion. Such failures can cause confusion regarding the effectiveness of the test cases. Hence, this step within *MuTAP* is dedicated to repairing the intended behavior of assertion oracles in the test cases (referred to as *IntendedBehaviorFixer* in Algorithm 3).

For each test case, *MuTAP* executes the *PUT* using the test inputs and compares the *PUT* return output to the assertion output. If the *PUT* return output matches the assertion output in the oracle, *MuTAP* deems it a correct assertion oracle with the intended behavior. Otherwise, it repairs these assertions by substituting the assertion output with the expected output from the *PUT* (Lines 22-27 in Algorithm 3). *MuTAP* excludes assertions for which the input types are incompatible with the *PUT*, for example, if the *PUT* expects a `list` of integers but the test input is a `string`. The result of this step is denoted as the *Initial Unit Test (IUT)*, which is a set of test cases generated by LLMC after refining, as depicted in ② in Figure 5.2.

5.3.3 Mutation Testing

Mutation Testing (MT) is used to evaluate the quality and effectiveness of test cases. Mutants are created by introducing artificial defects into the *PUT* to simulate faults. When a test case fails in the presence of a mutant, we classify it as a *killed mutant*. Conversely, if a test case succeeds, it is deemed a *surviving mutant*, indicating that the test cases within the unit test fail to detect it. The existence of surviving mutants highlights the limitations of the test cases, suggesting the need for either adding new test cases or improving existing ones. The *Mutation Score (MS)* quantifies the effectiveness of test cases by calculating the ratio of killed mutants to all mutants for a *PUT*.

Algorithm 4 provides a detailed description of this process. Drawing inspiration from [111], *MuTAP* employs MutPy [203] to generate diverse mutants for each *PUT* and calculate the *MS* (Lines 3-7 in Algorithm 4). Executing test cases on each mutant involves some preliminary setup. To facilitate this, *MuTAP* utilizes Python's built-in "setuptools.find_packages" to locate and install required packages such as "math", "NumPy", "pandas", "pytest", and

Algorithm 3: Refining

```
   Input: raw_IUT, PUT
   Output: IUT // The Initial Unit Test after refining
1  syntax_fixed_IUT ← SyntaxFixer (raw_IUT)
2  IUT ← IntendedBehaviorFixer (syntax_fixed_IUT, PUT)
3  return IUT
4
5  Procedure SyntaxFixer(raw_IUT)
6      if not AST.parse(raw_IUT) then
7          syntax_fixed_prompt ← CONCAT (INS_fix, raw_IUT)
8          syntax_fixed_IUT ← LLMC (syntax_fixed_prompt)
9      end
10     syntax_fixed_IUT ← SyntaxCheck (syntax_fixed_IUT)
11     return syntax_fixed_IUT
12
13 Procedure SyntaxCheck(syntax_fixed_IUT)
14     if AST.parse(syntax_fixed_IUT) then
15         return syntax_fixed_IUT
16     else
17         return SyntaxCheck (syntax_fixed_IUT [:error_line])
18     end
19
20 Procedure IntendedBehaviorFixer(syntax_fixed_IUT, PUT)
21     fixed_IUT ← {}
22     foreach test_case ∈ syntax_fixed_IUT do
23         expected_output ← PUT(test_case.input)
24         if expected_output ≠ test_case.output then
25             test_case.output ← expected_output
26         fixed_IUT.append(test_case)
27     end
28     return fixed_IUT
```

others. Furthermore, *MuTAP* implements setup functions responsible for creating temporary directories, which are utilized during the execution of test cases on the mutants. Subsequent to executing the test cases on the mutants and computing the *MS*, *MuTAP* properly tears down the setup by removing the temporary directories.

As shown in Lines 5-9 of Algorithm 1, if the *MS* for a *PUT* reaches 100%, *MuTAP* proceeds to the test case oracle minimization step (Subsection 5.3.5). Alternatively, if there are surviving mutants, *MuTAP* collects a list of these mutants and transfers them to the prompt augmentation step (Subsection 5.3.4).

5.3.4 Prompt Augmentation

Algorithm 5 shows the details of this step. If there is any surviving mutant from the previous step, *MuTAP* augments the initial prompt, *zero-shot* or *few-shot*, by incorporating four new components (Line 3 in Algorithm 5). The first component is *IUT*, the initial unit test generated by LLMC after refinement. The second component is an instruction in a natural language named INS_3 that clarifies the shortcoming of *IUT* by *"The test function, test(), cannot detect the fault in the following code"*. The third component is one of the surviving mutants of the *PUT*, named *SM*. The last component, INS_4 is an instruction in natural and

Algorithm 4: MutationTesting

```
   Input: PUT, IUT
   Output: MS, surviving_mutant
1  mutants ← MutPy(PUT)
2  surviving_mutant ← {}
3  foreach MUT ∈ mutants do
4  |   if exec(MUT, IUT) then
5  |   |   surviving_mutant.append(MUT)
6  |   end
7  end
8  MS ← (#(mutants) − #(surviving_mutant))/#(mutants)
9  return MS, surviving_mutant
```

programming language: the natural language context clarifies the task by asking to *"Provide a new test case to detect the fault in prior code"* and the programming language context acts only as a hint to guide LLMC for generating the output. An example is shown by ③ in Figure 5.2.

MuTAP re-prompt LLMC and repeats the refining step on the generated output (Line 4-5 in Algorithm 5). Then, it appends new generated test cases to the *IUT* that we call Augmented Unit Test (*AUT*). The *AUT* is passed to the *MT* step (Line 7 in Algorithm 5). *MuTAP* recursively repeats prompt augmentation till either the final test cases kill all the mutants (*MS = 100%*) or there is no surviving mutant that is not used in the augmentation process (Line 8 in Algorithm 5). An example of updating the mutant component in Figure 5.2, ③ is changed to ③′ by replacing SM_0 with SM_1. The ④′ indicates the generated test cases with LLMC after iterating the process on the next surviving mutant.

5.3.5 Oracle Minimization

The test cases generated by the LLMC usually consist of redundant assertions. Also, the augmentation process might introduce further redundant test cases. Presenting all of them (with redundancy) as the final output can cause confusion for developers. In the final step, similar to previous tools that generate mutation-driven test oracles [23, 204], *MuTAP* minimizes the number of assertions by utilizing a Greedy technique to eliminate the redundant assertions that do not improve the MS. This step is presented in Algorithm 6. *MuTAP* starts by tracking the number of mutants that each assertion kills and then chooses the test case containing the assertion that kills the maximum number of mutants. This process is then repeated by adding the test cases containing the next assertions that detect the most mutants (Line 4-10 in Algorithm 6). If adding this new assertion increases the MS, *MuTAP* keep the test case and its assertion. Otherwise, the test case will be discarded as redundant.

Algorithm 5: AugmentingPrompt

```
   Input: MS, PUT, initial_prompt, IUT, surviving_mutant
   Output: AUT // Augmented Unit Test
 1 if MS < 100% or surviving_mutant ≠ {} then
 2 |   SM ← surviving_mutant.pop()
 3 |   augmented_prompt ← CONCAT (initial_prompt, IUT, INS3, SM, INS4)
 4 |   raw_AUT ← LLMC (augmented_prompt)
 5 |   fixed_AUT ← Refining (raw_AUT, PUT)
 6 |   AUT ← IUT.append(fixed_AUT)
 7 |   MS ← MutationTesting (AUT, PUT)
 8 |   return AugmentingPrompt (MS, PUT, initial_prompt, AUT, surviving_mutant)
 9 else
10 |   return AUT
11 end
```

Algorithm 6: OracleMinimization

```
   Input: PUT, AUT
   Output: FUT
 1 MS_old ← 0
 2 FUT ← {}
 3 sorted_AUT ← sort(AUT, Descending= True) // sort each test case in AUT based on the MS
 4 foreach test_case ∈ sorted_AUT do
 5 |   FUT.append(test_case)
 6 |   MS ← MutationTesting (FUT, PUT)
 7 |   if MS > MS_old then
 8 |   |   MS_old ← MS
 9 |   else
10 |   |   FUT.delete(test_case)
11 |   end
12 end
13 return FUT
```

5.4 Evaluation

In this section, we describe the evaluations we designed and conducted to investigate the following research questions:

RQ1 How effective are test cases generated by *MuTAP* in comparison to test cases generated by automatic test generation tools in detecting bugs?

RQ2 What is the outcome of different components within *MuTAP*?

RQ3 How does the performance of *MuTAP* vary for each mutation type (artificial bugs)?

5.4.1 Experimental Setup

In this section, we present our experiment setup. Specifically, we describe the automatic test generation tool used to compare our results, clarify the LLMC of *MuTAP*, and indicate the baselines and benchmark datasets used in our experiments.

We conducted the experiment on the Cedar cluster of Compute Canada, which offers 32 cores

CPU, 1TB storage, and one v100l GPU with 32GB GPU Memory, and on a system running Linux 5.15.0-69-generic with AMD FX(tm)-6300 Six-Cores CPU, 512GB storage, and 16GB Memory.

Experimental Parameters

We call the initial prompt, *zero-shot* or *few-shot*, on LLMC up to 10 times and collect the outputs that meet two criteria as candidate test cases: the candidate should consist of two keywords of *assert* and the *function name* of *PUT*. If, after 10 runs, LLMC is not able to generate an output that contains those two keywords, we consider the task as a problematic task or a task for which *MuTAP* is not able to generate a test case.

Regarding the syntax fixing step, we run the syntax fixing prompt on the LLMC for up to 10 runs. If the syntax error remains unresolved even after 10 iterations, *MuTAP* employs the Python parser to locate the erroneous line. It then retains the lines preceding the buggy line, ensuring their freedom from syntax errors. If the removal of lines results in the absence of any remaining test cases (all test cases prove non-compilable), we classify the task as problematic.

Comparable Tool

Pynguin [111] is a well-known fully-automated test generation tool for a dynamically typed programming language such as Python. It uses different search-based algorithms to satisfy code coverage criteria, i.e., branch coverage. Pynguin first takes a Python code (method, module, etc.) as input and collects its information such as variable types, method names, and dependencies. Then it uses one of the search-based test generation algorithms (MIO [205], MOSA [206], DynaMOSA [207], etc.) to optimize test coverage and generate test cases. It randomly mutates (deletes, inserts, replaces) different values and statements within the test case to generate new test cases and executes them over the *PUT* to ensure their correctness. Finally, it generates assertions for test cases using a MT engine [111].

For our experiments, we employ Pynguin 0.17.0. with the DynaMOSA [207]. According to the evaluation of Pynguin [207], DynaMOSA shows the best performance compared to the other algorithms in generating test cases with this tool. We set the timeout of test generation to 600 seconds which is the default setting of the tool.

Large Language Model Component (LLMC)

We employ two different LLMs as the LLMC of *MuTAP*. The first one is OpenAI's Codex, designed specifically for code generation tasks [56]. We use *Code-davinci-002*, with a temperature of 0.8. The lower temperature causes less variation in the outputs of the model while the higher temperature increases the variation of output and then the chance of generating useful test cases over different iterations. The evaluation of CODAMOSA [22] shows that 0.8 is a reasonable temperature to generate useful test cases with Codex.

The second LLM is Meta's *llama-2-chat*, which has been iteratively refined using Reinforcement Learning with Human Feedback (RLHF) and is appropriate for dialog use cases [41]. Similar to Codex, we have configured the model's temperature to be 0.8. Furthermore, the model provides three distinct roles within the prompt: *system, user,* and *assistant*. These roles serve the purpose of clarifying each component of the prompt to the model by assigning specific components to each role. Different combinations of these roles can be utilized in each prompt to tailor the interaction with the model according to the specific requirements [41].

In our experiments, the role of the *system* is defined as *{You are a Python coding assistant. Always answer with Python code.}*, for all types of prompts, including *zero-shot*, *few-shot*, and *augmented* prompts. To handle the *zero-shot* prompt, we only set the *user's* role content to be a concatenation of *(INS_1, PUT_i, INS_2)*. For the *few-shot* prompt, we define the content of the *assistant* role as a set of demonstrative examples of Method (M) and Unit Test (UT), while the *user* role content is set to PUT_i. As for the *augmented* prompt, its various components are set up as follows:

{user: Initial Prompt,
assistant: *IUT*,
user: concat(*INS_3*, *SM_i*, *INS_4*)}

For both LLMs, the maximum number of generated tokens is set to 250 for generating test cases and 20 tokens for syntax fixing, based on previous studies on similar tasks [22, 208]. The stop word is defined as *quote (")* for *zero-shot* and as $< /test >$ for *few-shot* prompt. For the rest of the hyperparameters, we keep the model's default values.

To avoid overfitting on the benchmarks data, *MuTAP* repeats all prompts on Codex or llama-2-chat for up to 10 runs. If after 10 runs, the requirement for generating test cases is not satisfied, *MuTAP* considers it as a problematic or unsolved task.

It is important to note that *MuTAP* is not limited to these two models, and its LLMC can be replaced with any other LLM as required.

Baselines

In addition to Pynguin, we propose two baselines for each LLM to evaluate our proposed method, *MuTAP*.

Before-refining: The first baseline is the output of the initial prompt on LLMC (Codex or llama-2-chat), without fixing syntax errors or repairing the intended behavior. Since assertions with unintended return values can fail on mutants or buggy code and present invalid effectiveness (False Positive), we omit those assertions in this baseline to avoid this side effect. If the output of the model has syntax errors, we consider it as a wrong test and consequently consider the task as a problematic or unsolved task.

After-refining: The second baseline is the output of the initial prompt on LLMC (Codex or llama-2-chat), after applying the following steps: *Refining* (Subsection 5.3.2) and *Oracle Minimization* (Subsection 5.3.5).

Mutant Generator

To apply MT, we need to generate different mutant versions of a PUT by injecting bugs into its different lines. For this purpose, we use *MutPy* version 2.0 [203]. *MutPy* is a MT tool for code in Python 3.3+. It benefits from different mutation operators to generate the mutants. The list of mutation operators used in our experiment with corresponding examples is shown in Table 5.1. *MutPy* injects one operator at a time to generate the mutant if the operator is applicable on PUT.

Benchmark Datasets

To conduct our experiments, we use two different benchmarks. The first one is *HumanEval* [56] which is a benchmark to evaluate LLMs that generate code. It has 164 human-written programming problems at easy to medium levels. Each problem has different attributes such as descriptions and reference solutions. We use the reference solution of each task as a PUT.

The second one, *Refactory* [69], is a benchmark for Python bug repairing [165]. It has 1710 buggy students' submissions for 5 assignments of a Python programming course. Each assignment has a correct reference solution that we use as PUT. The advantage of this dataset is buggy code snippets generated by humans that give us the opportunity to evaluate test cases generated by *MuTAP* on real bugs and compare them with Pynguin and our baselines.

Table 5.1 List of the mutation operators in our experiments used by *MutPy* sorted by alphabetical order.

Operator	Example	Mutant
AOD - arithmetic operator deletion	result.append(numbers[-1])	result.append(numbers[1])
AOR - arithmetic operator replacement	return number % 1.0	return number * 1.0
ASR - assignment operator replacement	current_depth += 1	current_depth -= 1
BCR - break continue replacement	if i % j != 0: break	if i % j != 0: continue
COD - conditional operator deletion	if not string: return ' '	if string: return ' '
COI - conditional operator insertion	if balance < 0: return True	if (not balance < 0): return True
EHD - exception handler deletion	except: pass	except: raise
EXS - exception swallowing	except: return False	except: pass
LCR - logical connector replacement	if s[-1] == 'y' or s[-1] == 'Y':	if s[-1] == 'y' and s[-1] == 'Y':
ROR - relational operator replacement	if c[n] <= 1:	if c[n] >= 1:
SIR - slice index remove	l[::3] = sorted(l[::3])	l[::3] = sorted(l[:])

5.4.2 Experimental Results

In this section, we discuss our findings for each RQ.

RQ1: How effective are test cases generated by *MuTAP* in comparison to test cases generated by automatic test generation tools in detecting bugs?

Since our study focuses on MT to improve the effectiveness of test cases, we compare *MuTAP* with Pynguin and our baselines in terms of MS, number of killed mutants, and number of *PUT* with 100% MS. MS (Avg.)$\pm$*std* presents the average of MS with its standard deviation over all PUTs in the HumanEval benchmark. It is worth mentioning that we only consider *PUT*s with correct test cases to calculate the average MS for each method. For this reason, we report the total number of killed mutants and the total number of *PUT*s with 100% MS for a fair comparison.

Table 5.2 shows the obtained results for the *HumanEval* benchmark. Prior to syntax fixing and intended behavior repair (*before-refining*), the test cases generated by Codex and llama-2-chat are incorrect for 73 and 68 (out of 164) *PUT*s, respectively, when using the *zero-shot* initial prompt. However, they manage to kill 295 and 318 mutants (out of 1260), respectively.

The initial prompt has a more pronounced impact on the output of Codex compared to llama-2-chat. Switching the initial prompt to *few-shot* decreases the number of *PUT*s without test cases to 39, while also raising the number of killed mutants to 508 when using Codex as LLMC. On the other hand, when using llama-2-chat, the number of *PUT*s without test cases reduces to 60, and the number of killed mutants increases from 318 to 325. This difference in performance could be attributed to llama-2-chat being more suitable for dialog prompts, and

Table 5.2 Evaluation result of test cases generated by *MuTAP* on *synthetic* buggy programs.

Prompt	Model	Method	# Test Cases (avg)	# Problematic PUT (out of 164)	MS (avg) (%) $\pm std$	# Killed Mut (out of 1260)	Task MS=100% (out of 164)
-	-	Pynguin	1.5 (min=1, max=4)	31	$65.94\% \pm 30.78$	649	28.22% (46)
Zero-shot	**Codex**	Before-refining	1.5 (min=1, max=3)	73	$72.15\% \pm 26.95$	296	11.04% (18)
		after-refining	2.1 (min=1, max=3)	30	$76.82\% \pm 24.35$	749	24.54% (40)
		MuTAP	**2.5 (min=1, max=4)**	**30**	$89.13\% \pm 20.32$	**869**	**41.72% (68)**
Zero-shot	**llama2-chat**	Before-refining	1.2 (min=1, max=3)	68	$62.60\% \pm 28.82$	318	17.79% (29)
		After-refining	2.2 (min=1, max=5)	0	$84.04\% \pm 17.41$	1059	53.98% (88)
		MuTAP	**2.5 (min=1, max=5)**	**0**	$91.98\% \pm 13.03$	**1159**	**68.09% (111)**
Few-shot	**Codex**	Before-refining	1.5 (min=1, max=3)	39	$72.68\% \pm 26.23$	508	15.95% (26)
		After-refining	2.2 (min=1, max=3)	27	$82.73\% \pm 21.91$	829	34.97% (57)
		MuTAP	**2.6 (min=1, max=7)**	**27**	$92.02\% \pm 13.55$	**922**	**49.69% (81)**
Few-shot	**llama2-chat**	Before-refining	1.5 (min=1, max=3)	60	$64.51\% \pm 24.11$	325	22.69% (37)
		After-refining	2.5 (min=1, max=5)	0	$85.16\% \pm 16.36$	1073	75.05% (93)
		MuTAP	**2.6 (min=1, max=7)**	**0**	$93.57\% \pm 11.18$	**1179**	**69.93% (114)**

using a prompt with a pair of demonstrative input and output, devoid of natural language context, does not improve the model's performance significantly.

In contrast, Pynguin, as the state-of-the-art automatic test generation tool, outperforms the output of both LLMs, before-refining, by killing 649 mutants and failing to generate test cases for 31 tasks.

After applying the post-processing steps of syntax fixing and intended behavior repair, *MuTAP* with both LLMs perform better than Pynguin in terms of killing more mutants. Notably, when using both *zero-shot* and *few-shot* prompts, llama-2-chat is able to generate correct test cases for all *PUT*s, after-refining. However, their effectiveness in terms of killing mutants is measured at 84.04% and 85.16% with the *zero-shot* and *few-shot* prompts, respectively.

On the other hand, the MS of test cases generated by Codex after refining is 76.82% and 82.73% with the *zero-shot* and *few-shot* prompts, respectively. Despite this improvement, Codex still fails in generating correct test cases for 30 (with *zero-shot*) and 27 (with *few-shot*) *PUT*s after refining.

MuTAP, enhances the effectiveness of test cases generated by both LLMs, Codex and llama-2-chat, achieving MS of 89.13% and 91.98% with the *zero-shot* prompt, and an MS of 92.02% and 93.57% with the *few-shot* prompt, respectively. Particularly, *MuTAP* with the *few-shot* prompt when using llama-2-chat as its LLMC manages to kill 1179 mutants out of 1260 and generates test cases with MS=100% for up to 70% of *PUT*s, demonstrating a remarkable improvement in the effectiveness of test cases compared to the Pynguin with 649 killed mutants and 28.22% *PUT*s with MS=100%.

As we report the average of MS across all PUTs in the *HumanEval* benchmark in Table 5.2,

Table 5.3 p-value of Mann–Whitney U test between *MuTAP* and comparable methods

MuTAP		Avg. MS	
Prompt	Model	Method	p-value
Zero-shot	llama-2-chat	Pynguin	<< .001
		Before-refining	<< .001
		After-refining	0.003
Few-shot	llama-2-chat	Pynguin	<< .001
		Before-refining	<< .001
		After-refining	0.002
Zero-shot	Codex	Pynguin	<< .001
		Before-refining	<< .001
		After-refining	< .001
Few-shot	Codex	Pynguin	<< .001
		Before-refining	<< .001
		After-refining	< .001

we also conduct a statistical analysis to determine whether the effectiveness of test cases generated by *MuTAP* is significantly different from those generated by Pynguin and other baselines. We employ the Mann-Whitney U-test with a p-value threshold of 5% (0.05) for this comparison. In terms of MS, the null hypothesis states that the test cases generated by *MuTAP* are not significantly more effective than those generated by other comparable methods. The alternative hypothesis suggests that the test cases generated by *MuTAP* are more effective than those generated by comparable methods. Table 5.3 presents the results of this test. Since all p-values are below 0.05, the null hypothesis is rejected, showing statistically significant superiority of *MuTAP*. Across different setups of *MuTAP* with various LLMC, Codex or llama-2-chat, and with different initial prompts, either *Zero-shot* or *Few-shot*, the test cases generated by *MuTAP* for different PUTs are more effective than those generated by Pynguin and also baselines (Before-Refining and After-refining) methods with a 95% confidence level (p-value = 5%).

To evaluate the performance of *MuTAP* on real buggy programs, we employ the *Refactory* benchmark. Table 5.4 that shows the results of this dataset confirms our findings on *HumanEval*. To evaluate *MuTAP* on real buggy code, we apply the following steps. First, we generate the mutants of each *PUT* in this dataset. Second, we conduct the prompt augmentation process and finalize the test cases for each *PUT*. Then, we apply test cases generated by *MuTAP* on students' buggy code in *Refactory*, followed by test cases generated by Pynguin and LLMs *After-refining*, to assess the effectiveness of test cases generated by different

methods in detecting buggy code.

MuTAP with *few-shot* learning while using llama-2-chat as its LLMC identifies 468 more buggy code compared to Pynguin (with an MS of 94.91% vs. 67.54%) and 111 more buggy code compared to *After-refining* (with an MS of 94.91% vs. 82.51%). Furthermore, *MuTAP* discovers 79 buggy code that were not detected by either Pynguin or llama-2-chat's test cases *After-refining* process. When using Codex, *MuTAP* detects 73 buggy code that were missed by both Pynguin and Codex's test cases *After-refining* stage. Moreover, *MuTAP* excels in generating more effective test cases, with an average of 2.6 test cases after applying greedy optimization.

Overall, *MuTAP* using both llama-2-chat and Codex demonstrates better performance compared to Pynguin in terms of killing mutants and detecting buggy code. The effectiveness of these test cases in detecting defects is improved through post-processing steps of refining and prompt augmentation.

Table 5.4 Evaluation results on *real* buggy programs.

Prompt	Model	Method	# Test Cases (avg)	Bug Detected (out of 1710)
-	-	Pynguin	1.25 (min=1, max=4)	67.54% (1155)
Zero-shot	**Codex**	After-refining	1.2 (min=1, max=2)	79.87% (1356)
		MuTAP	**1.6 (min=1, max=3)**	**84.03% (1437)**
Zero-shot	**llama-2-chat**	After-refining	1.2 (min=1, max=3)	86.43% (1478)
		MuTAP	**2.2 (min=1, max=4)**	**93.22% (1594)**
Few-shot	**Codex**	After-refining	1.6 (min=1, max=3)	82.51% (1411)
		MuTAP	**2.2 (min=1, max=4)**	**89.41% (1529)**
Few-shot	**llama-2-chat**	After-refining	2.1 (min=1, max=4)	88.42% (1512)
		MuTAP	**2.2 (min=1, max=4)**	**94.91% (1623)**

Finding 1: *MuTAP* generates more effective test cases compared to Pynguin and conventional *zero-shot* and *few-shot* learning on LLM. The number of *MuTAP*'s test cases is not much greater than the output of other methods after minimization. Additionally, LLM with dialog setup performs better on the augmented prompt. In conclusion, the effectiveness of LLM-generated test cases can be enhanced through prompt augmentation using surviving mutants and post-processing refinement.

RQ2: What is the outcome of different components within *MuTAP*?

Syntax Fixer: On average, the percentage of test cases with syntax errors is 38.98% and 26.48% when using the *zero-shot* and *few-shot* prompts, respectively, with Codex. When

employing llama-2-chat, this percentage is 33.85% and 26.32% with the *zero-shot* and *few-shot* prompts, respectively.

When considering syntax errors, three factors contribute to decreasing them in the output of LLMs. The first factor is the type of initial prompt. As shown in Table 5.5 on the *HumanEval* benchmark, *few-shot* learning results in fewer syntax errors in the output of both LLMs. Specifically, when using Codex, the percentage of syntax errors decreases from 44.79% to 29.03% after-refining, and for *MuTAP*, it decreases from 33.17% to 23.93%. With llama-2-chat as the LLMC, the percentage of syntax errors decreases from 38.03% to 26.99% after refining, and from 29.66% to 25.64% for *MuTAP*.

Table 5.5 Syntax error fixing of test cases. The syntax Error Rate shows the ratio of unit tests with syntax errors.

Model	Method	Prompt	# Run (avg)	Syntax Error Rate	Fixed by Model	Fixed by Omitting Lines
Codex	After-refining	Zero-shot	9.1	44.79%	16.44%	60.27%
		Few-shot	9.5	29.03%	12.96%	83.33%
	MuTAP	Zero-shot	9.7	33.17%	16.18%	79.41%
		Few-shot	9.5	23.93%	12.82%	84.62%
llama2-chat	After-refining	Zero-shot	7.1	38.03%	30.64%	63.86%
		Few-shot	6.8	26.99%	31.81%	57.96%
	MuTAP	Zero-shot	6.9	29.66%	32.17%	61.05%
		Few-shot	6.8	25.64%	32.45%	60.40%

The second impactful factor, which is also the primary factor, is the *Syntax Fixing* component. As shown in Table 5.5, when using Codex, this component in *MuTAP* on average fixes 14.5% of syntax errors by utilizing the LLMC and addresses 81.37% of syntax errors by omitting the lines causing the errors. On the other hand, when using llama-2-chat as the LLMC of *MuTAP*, the *Syntax Fixing* component, on average, resolves 32.31% of syntax errors through re-prompting the LLMC, and 60.73% of the errors by omitting the problematic lines.

The final factor contributing to the improvement of syntax errors in test cases is the prompt augmentation process in *MuTAP*. By augmenting the prompt with *IUT*, the occurrence of syntax errors in the output of Codex with the *zero-shot* technique decreases from 44.79% to 33.17%. Similarly, with llama-2-chat and the *zero-shot* prompt, the percentage of syntax errors reduces from 38.03% to 29.66%. Augmenting the prompt with *IUT* provides illustrative examples of test cases and serves a similar purpose to the demonstrative examples in the *few-shot* learning prompt, effectively reducing syntax errors in the output of LLMs.

Our finding on the *Refactory* benchmark shows *MuTAP* generates test cases with syntax errors in only one *PUT* (out of 5) using Codex and *zero-shot* learning. Moreover, none of those syntax errors could be fixed by re-prompting LLMC. On the other hand, for both initial prompt types, syntax errors decrease to zero using llama-2-chat.

Intended Behavior Repair: In the case of repairing intended behavior, two distinct

factors contribute to reducing the error rate in assertion oracles. As shown in Table 5.6, the *Intended Behavior Repair* step, when using Codex as the LLMC, on average, fixes 83.98% and 89.86% of incorrect behaviors in the *after-refining* and *MuTAP*, respectively. When utilizing llama-2-chat, this step repairs 84.35% and 95.96% of unintended behavior in the *after-refining* and *MuTAP*, respectively.

In addition to the *Intended Behavior Repair* step, the prompt augmentation step in *MuTAP* significantly reduces the occurrence of unintended behavior in test cases. For instance, when using Codex with a *zero-shot* prompt, the assertions with unintended behavior, such as wrong return values, decrease from 63.63% to 19.38%. Similarly, with llama-2-chat and using *few-shot* prompt, the assertions with unintended behavior decrease from 63.25% to 10.75%. The reason behind this improvement could be attributed to the usage of *IUT*s (Initial Unit Tests) in *MuTAP* for augmenting the initial prompt. These *IUT*s already represent the intended behavior of the *PUT*, thereby assisting the LLM in suggesting test cases with less unintended behavior (i.e., fewer wrong return values). Also, on the *Refactory* benchmark, *MuTAP* repaired all assertions with incorrect behavior on the output of augmented prompts.

Unlike syntax errors, the prompt type does not significantly help with unintended behavior in assertions. The combination of the *Intended Behavior Repair* step and the prompt augmentation process improves the effectiveness of test cases, ensuring that they align with the intended behavior of *PUT*.

Surviving Mutants Representation: We also investigated the impact of surviving mutants' order on MS during prompt augmentation. Figure 5.3 illustrates the effect of augmenting the prompt with a random order of surviving mutants over 5 runs for all *PUT*s. For this comparison, we randomly selected one of the surviving mutants of each *PUT* with $MS < 100\%$ and utilized it to augment the initial prompt. We then calculated the average MS for all *PUT*s. Subsequently, we randomly chose the second surviving mutant for the remaining *PUT*s with MS $< 100\%$ (if any), repeated the augmentation process as a second iteration, and calculated the average MS for all *PUT*s again. We continue to iterate this process (axis in Figure 5.3) until either there are no more *PUT*s with MS $< 100\%$ or no more surviving mutant that is not utilized in the argumentation process.

As shown in Figure 5.3, each data point represents an iteration of the augmentation step and the average MS for all *PUT*s across five runs, derived from a random selection of surviving mutants. The shaded area illustrates the standard error of the average MS across these five runs in each iteration. The results show that the standard error over 5 runs in each iteration is not significant. However, during the initial iterations (up to 7 iterations), the standard error around the average MS over five runs is greater than what is observed in the final

iterations. That is because after several iterations over the augmentation step with different surviving mutants, the improvement in MS stalls. Notably, more than 90% of the MS is achieved by using only half of the surviving mutants, and the improvement in MS stalls after a certain iteration of the augmentation step for different LLMs. For example, when using Codex as LLMC, in *zero-shot* learning, the MS stops improving even though, on average, 27 surviving mutants (out of 226) are not utilized in the prompt augmentation step. Similarly, in *few-shot* learning, this number is equal to 24 (out of 106).

Our results for RQ2 demonstrate that test cases generated by LLMs, regardless of the prompt type, require post-processing, such as syntax correction or intended behavior repair, in order to function properly and detect bugs effectively. Also, the order of surviving mutants to augment the prompt does not significantly impact the MS gain.

Finding 2: The *Syntax Fixing* and *Intended Behavior Repair* fix up to 95.94% and 89.86% of syntax and functional errors in test cases, respectively. The prompt augmentation in *MuTAP* decreases the unintended behavior in the output of LLMs significantly (44.36% using Codex and 52.5% using llama-2-chat). Furthermore, only a small number of mutants (up to 27) do not contribute to the improvement of MS.

RQ3: How does the performance of *MuTAP* vary for each mutation type (artificial bugs)?

In this RQ, we evaluate the performance of *MuTAP* in different mutant types. We report the total number and number of killed mutants by each method on the *HumanEval* benchmark in Table 5.7. The performance of all techniques per mutant type is reported to help the comparison as well. The total number of mutants in each type is different for each method since the number of problematic *PUT*s is not the same for all methods. The MS for each type/method indicates the ratio of killed mutants out of the total number of mutants in that type. There are some mutant types that are more common (more samples in those types) such as *AOR*, *COI*, and *ROR* (an example for each mutant type is shown in Table 5.1). The number of mutants in each type depends on the *PUT*. For example, in the *HumanEval*, there are few *PUT*s with exception handling. Consequently, there are few mutants in the *EHD*.

In general, *MuTAP* shows better or similar performance in all mutant types compared to Pynguin and the output of LLMs *After-refining* of both LLMs. Considering *ASR* as an example, *MuTAP* shows the highest performance on this mutant type among all methods. For example, test cases generated by Pynguin identified 45 mutants in this category while test cases generated by *MuTAP* using llama-2-chat and the *few-shot* prompt identified 79

Table 5.6 Evaluation results of *Intended Behavior Repair*. The Assertion Error Rate shows the ratio of assertions with wrong behavior.

Model	Method	Prompt	Assertion Error Rate	Repaired	Not Repaired
Codex	After-refining	**Zero-shot**	63.63%	82.21%	17.79%
		Few-shot	62.84%	85.75%	14.25%
	MuTAP	**Zero-shot**	19.38%	89.71%	10.29%
		Few-shot	18.36%	90.00%	10.71%
llama-2-chat	After-refining	**Zero-shot**	60.27%	81.80%	18.19%
		Few-shot	63.25%	86.90%	13.09%
	MuTAP	**Zero-shot**	23.40%	94.06%	5.94%
		Few-shot	10.75%	94.91%	5.09%

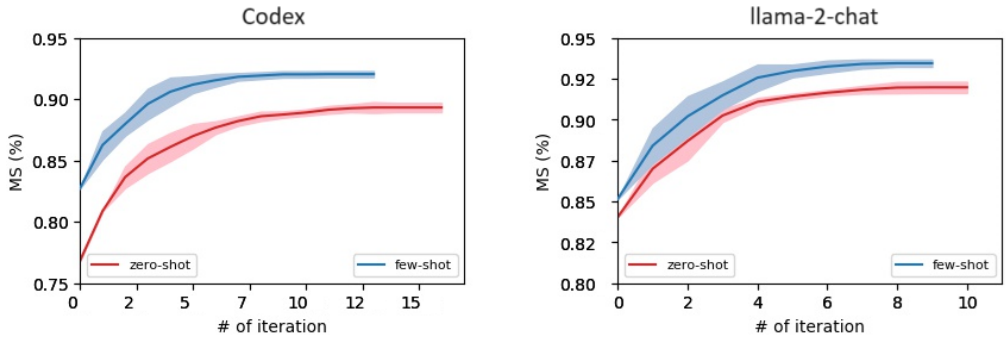


Figure 5.3 **The impact of using surviving mutants in different random orders on the MS.** Results are averaged over 5 runs, the line shows the mean and the shaded area shows standard error. Each data point represents the average MS for all PUTs across five different runs in an iteration, wherein the surviving mutants were randomly selected for the prompt augmentation process.

mutants in this category (out of 84).

For one of the mutant types, *BCR*, which is a rare type in our benchmarks, *MuTAP* and *After-refining* with both *zero-shot* and *few-shot* initial prompt, along with using Codex, show the same performance. However, when employing llama-2-chat, *MuTAP* outperforms the others by killing more mutants of this type. For another rare type of mutant in our dataset, *EHD*, it is noteworthy that Codex, despite using both initial prompt types and the augmentation process, fails to generate test cases to detect the two mutants present in this category. In contrast, *MuTAP* with the *few-shot* initial prompt and llama-2-chat successfully killed all of the mutants in this category.

Table 5.7 Evaluation of killed mutants for each type of injected operator into PUTs.

	Pynguin		Zero-shot								Few-shot							
			Codex				llama-2-chat				Codex				llama-2-chat			
			After-refining		*MuTAP*		After-refining		*MuTAP*		After-refining		*MuTAP*		After-refining		*MuTAP*	
Type	**killed**	**total**	**killed**	**total**	**killed**	**total**	**killed**	**total**	**killed**	**total**	**killed**	**total**	**killed**	**total**	**killed**	**total**	**killed**	**total**
AOD	13 (39.39%)	33	20 (62.50%)	32	28 (87.50%)	32	36 (80.00%)	45	39 (86.67%)	45	27 (79.41%)	34	32 (94.12%)	34	37 (82.22%)	45	40 (88.89%)	45
AOR	248 (67.39%)	368	274 (74.66%)	367	336 (91.55%)	367	390 (87.05%)	448	410 (91.52%)	448	290 (77.33%)	375	347 (92.53%)	375	394 (87.95%)	448	417 (93.08%)	448
ASR	45 (60.00%)	75	56 (74.67%)	75	60 (80.00%)	75	74 (88.10%)	84	79 (94.05%)	84	57 (76.00%)	75	64 (85.33%)	75	75 (89.29%)	84	79 (94.05%)	84
BCR	2 (40.00%)	5	2 (40.00%)	5	2 (40.00%)	5	5 (55.56%)	9	5 (55.56%)	9	2 (40.00%)	5	2 (40.00%)	5	5 (55.56%)	9	6 (66.67%)	9
COD	8 (53.33%)	15	12 (80.00%)	15	15 (100.00%)	15	15 (68.18%)	22	16 (72.73%)	22	15 (88.24%)	17	17 (100.00%)	17	15 (68.18%)	22	17 (77.27%)	22
COI	130 (81.76%)	159	145 (91.19%)	159	154 (96.86%)	159	194 (85.46%)	227	216 (95.15%)	227	161 (96.99%)	166	164 (98.80%)	166	200 (88.11%)	227	218 (96.04%)	227
EHD	1 (100.00%)	1	0 (0.00%)	0	0 (0.00%)	0	1 (50.00%)	2	2 (100.00%)	2	0 (0.00%)	1	1 (100.00%)	1	1 (50.00%)	2	2 (100.00%)	2
EXS	0 (0.00%)	0	0 (0.00%)	1	1 (100.00%)	1	1 (100.00%)	1	1 (100.00%)	1	0 (0.00%)	1	1 (100.00%)	1	1 (100.00%)	1	1 (100.00%)	1
LCR	14 (45.16%)	31	22 (70.97%)	31	23 (74.19%)	31	30 (69.77%)	43	37 (86.05%)	43	24 (72.73%)	33	27 (81.82%)	33	32 (74.42%)	43	39 (90.70%)	43
ROR	174 (66.67%)	261	200 (76.92%)	260	227 (87.31%)	260	281 (84.13%)	334	316 (94.61%)	334	227 (86.97%)	261	239 (91.57%)	261	282 (84.43%)	334	320 (95.81%)	334
SIR	10 (33.33%)	30	18 (60.00%)	30	23 (76.67%)	30	32 (71.11%)	45	38 (84.44%)	45	26 (76.47%)	34	28 (82.35%)	34	31 (68.89%)	45	40 (88.89%)	45
Total	645 (65.95%)	978	749 (76.82%)	975	869 (89.13%)	975	1059 (84.05%)	1260	1159 (91.98%)	1260	829 (82.73%)	1002	922 (92.02%)	1002	1073 (85.16%)	1260	1179 (93.57%)	1260

Finding 3: The test cases generated by *MuTAP* are equally or more effective in killing different types of mutants compared to those generated by Pynguin and the baseline method. Also, using an LLM with dialog setup can increase the number of killing mutants in different mutant types while applying prompt augmentation.

5.5 Discussion

MuTAP leverages the code synthesis capabilities of LLMs and employs prompt-based learning to assist developers in generating effective test cases without the need for the computationally expensive fine-tuning of LLMs.

The findings in this chapter highlight the significant impact of the refinement process on enhancing the reliability and effectiveness of test cases generated by LLMs. While the choice of the initial prompt type (zero-shot or few-shot) can reduce the number of problematic PUTs for which the LLMC fails to generate test cases, it has a limited effect on the overall test effectiveness in terms of mutant detection (improving MS) and revealing bugs.

MuTAP, with its post-processing steps including refinement and prompt augmentation, significantly improves MS and substantially increases the number of detected bugs. For example, it improves the count from 325 to 1,179 killed mutants when employing *llama2-chat* and few-shot initial prompts. This improvement is also observed in the detection of real bugs. MuTAP can identify 28% more faulty human-written code, while 17% of them remain undetected by both Pynguin and the *zero-shot/few-shot* learning technique using LLMs.

Listing 5.4 shows a sample test case generated by Pynguin for the *PUT* of our motivating example in Section 5.2.2. While Pynguin generates test inputs as random integers and mutates those values to generate new test cases, *MuTAP* relying on the power of its LLMC, produces test cases that are more natural-looking and correlated with input/output type and the functionality of the *PUT*.

Furthermore, adding the test cases generated by the LLMC after refinement as one of the augmentation components in the prompt reduces the syntax error rate and significantly decreases unintended behavior (unexpected test outputs within assertions) in the test cases.

While MuTAP generally enhances the effectiveness of test cases across various mutant types, there are specific mutant types for which test cases generated by the LLMC perform better. Conversely, there are mutant types for which augmenting the prompt with them does not yield significant improvements, such as BCR (Break Continue Replacement) mutants, which is also an uncommon type in our dataset.

Developers can use *MuTAP* to generate effective test cases in terms of bug detection, leveraging the capabilities of LLMs. Furthermore, *MuTAP* can be integrated into the test generation component of the GitHub Copilot lab [209], offering developers valuable suggestions for more effective test cases. Since the mutants can be generated automatically, prompt augmentation can be applied without human engagement.

Prompting

In the few-shot learning prompt, one of the illustrative examples was drawn from the Pynguin documentation, while another was sourced from the HumanEval dataset. It is noteworthy that this particular example was excluded from the evaluation applied to the HumanEval dataset. The effectiveness of few-shot learning relies on the relevance of the illustrative examples to the target task, and the format of these examples serves as a guide for the model in predicting the output format [210]. However, in our results, we did not investigate how modifying the examples in the few-shot prompt might impact the results. Instead, we adhered to leveraging these examples to guide the model in predicting the output format of test cases which contributes to a lower rate of syntax errors observed in test cases when employing few-shot learning as an initial prompt.

In the zero-shot initial prompt, one of the fixed instructions, INS_2, indicates a fixed function name for the test function. The primary reason behind using the name *"def test()"* in the initial Zero-shot prompt for the test function is twofold. Firstly, it serves as a hint to the model, indicating our expectation for a set of test cases organized within a function. Secondly, it facilitates the automation of experiments and provides an identifier or tag for automatically collecting the entire test function from the output generated by the model. Despite this, given that we already incorporate the PUT with its name in the initial prompt, it signals the model to generate a test function specifically for the mentioned PUT. In some instances, we have observed that the model (i.e., llama-2-chat) even replaced the simple test function name provided in the prompt with a more readable name in the form of "test_PUT_Name()",

such as “test_any_int()”. To evaluate the impact of allowing LLM to determine the function name, we randomly sampled 20% of the programming tasks from the HumanEval benchmark, resulting in 33 sample cases. We omitted the test function name from the initial zero-shot prompt in INS_2 and subsequently executed it using “llama-2-chat” for these sample PUTs and applied a Mann-Whitney U-test with a significance level of 5% (alpha=0.05). While the identifiers suggested by LLM as a function name are more readable and relevant to the functionality of PUTs, the result of the test showed no significant difference in the MS or the effectiveness of test cases following this modification in the initial prompt (MS = 86.22% and standard deviation = 19.34% on the sample-set).

To address syntax errors in test cases through re-prompting the LLMC, we used a fixed instruction (INS_fix), similar to the approach proposed by Zhang et al. [72], to leverage the LLM for syntax error fixing. If the model fails to fix the syntax error, MuTAP omits the line containing the error, based on the suggestion provided by Lemieux et al. [22]. This is because our observations indicate that the primary cause of syntax errors in the generated test cases by the model is often the last line of the test function when the model is unable to complete it. We observed that re-prompting the LLMC to address syntax errors sometimes resolves the error line by completing the last line of the test function but it introduces new lines, with one of them remaining incomplete. This primarily contributes to the lower rate of syntax error repairs by the LLM in our results. Consequently, we finalize our syntax fixing steps by omitting the error line (i.e., the incomplete line in the test functions).

In addition, for INS_3 and INS_4, the prompt serves as an instruction to call the LLMC to generate a test case capable of killing the surviving mutant in the prompt. As illustrated in Appendix B, it demonstrates the ability of LLMC to explain the differences between the surviving mutant and the PUT, pinpointing the specific line containing the bug, and subsequently producing a test case to detect that difference. However, including information about the buggy line of the mutant in the prompt could potentially enhance the effectiveness of the prompt augmentation step.

Execution Time

The open-access API of Codex has a limit on the number of requests (20 per minute) and the number of tokens (40,000 per minute). For this reason, our experiment needs to stop calling the API once in a while to not exceed the limit. As a result, we present the processing time analysis using llama-2-chat. The overall processing time of *MuTAP* on *HumanEval* dataset while using llama-2-chat is on average 39.75 seconds with zero-shot learning (with a min of 16.16 and a max of 56.66 seconds) and 42.11 seconds with the few-shot prompt (with a

```
def test_case_0():
    int_0 = -2973
    int_1 = 815
    bool_0 = module_0.any_int(int_0, int_0, int_1)
    assert bool_0 is False
```

Figure 5.4 A sample test case generated by Pynguin for the PUT in the motivation example presented in Figure 5.4.

min of 18.2 and a max of 64.2 seconds) per task. It includes on average building and calling initial prompts on LLMC with an average of 10.26 seconds, syntax fixing including calling the syntax fixing prompt on LLMC with 10.3 seconds, intended behavior repair at 0.38 seconds, MS calculation at 1.7 seconds, creating augmented prompts and calling them on LLM with 12.05 second and greedy optimization with 1.4 seconds. It is noteworthy that following the prompt augmentation step, *MuTAP* must reiterate the processes of syntax fixing, intended behavior repair, and greedy steps which are already included in the overall processing time. Among all steps of *MuTAP*, the most time-consuming ones are those that entail inferring output from the LLM. Conversely, the overall processing time on the same benchmark with Pynguin to complete searching the required space is on average 44.16 seconds with a min of 2.7 and a max of 10 minutes which is the default timeout of the tool.

The benefit of dialog LLM

Our findings indicate that the dialog setup of llama-2-chat provides *MuTAP* with the flexibility to assign distinct roles to each component of the augmented prompt. For instance, by assigning *IUT* to an assistant role during the prompt augmentation process, the likelihood of repeating the initial tests in the generated output is reduced, while the chance of generating new test cases for detecting surviving mutants is increased. An example of this prompt on llama-2-chat is illustrated in Appendix B.

Evalution Metrics

Prior studies [55, 116, 195] that involve the generation of assertions through LLMs have employed the *"exact match"* as one of their evaluation metrics. Exact match calculates the percentage of test cases generated by LLMs (the inferred output) that lexically match with the ground truth test cases (expected output). However, CIDAR [116] has already discussed the inadequacy of exact match as a suitable metric for assessing assertions produced by

LLMs. This reason is that the model often generates assertions that are semantically correct but may not precisely match the ground truth. In our study, *MuTAP* executed each test case including assertions, both on the *PUT* and on its mutants to assess their correctness and effectiveness, using MS. MS is a metric frequently used in prior studies and it serves as an effective metric for evaluating the quality of the test oracle [105]. While, in this paper, we focus on improving the effectiveness of test cases in terms of fault detection, there are other metrics such as test coverage that can assess other quality aspects of a test case. Improving MS does not necessarily lead to good coverage and test coverage is weakly correlated with the efficiency of tests in fault detection [196] and is challenged as a measure of test effectiveness in revealing faults [197, 198], which can make it challenging for our proposed method to perform well on both metrics [204, 211].

Furthermore, the results presented in [212] indicate that approximately 60% of the test cases generated by Codex encounter compilation issues due to syntax errors. The incorporation of syntax correction and intended behavior repair steps in our proposed method, *MuTAP*, significantly enhances the utility of the tests generated by LLMs.

Surviving mutants

We augment the prompt at each iteration for each *PUT* with a single surviving mutant. The average number of mutants for all *PUT*s in *HumanEval* and *Refactory* are 6.6 and 4.2 and the average number of surviving mutants are 3.6 and 1.8, respectively. Using a combination of surviving mutants to augment the prompt could impact the speed of reaching 100% MS. However, not all surviving mutants used in prompt augmentation contribute to improving MS, sometimes new test cases that address one mutant can also kill the remaining surviving mutants.

5.6 Threats to Validity

In this section, we will discuss the threats to the validity of our approach and results.

Internal validity

In this study, we employed two different prompt-based learning techniques: *zero-shot* and *few-shot*. However, we did not explore the potential impact of altering the natural language instructions or demonstrative examples (for *few-shot* learning) within our prompts. Modifying these instructions or utilizing different demonstrative examples more closely aligned with the *PUT*'s functionality could potentially enhance the results. Conversely, using, for

example, lengthy natural language instructions might potentially have an adverse effect on the results.

To repair syntax errors in test cases through re-prompting the LLMC, we have employed the approach presented in [72]. We did not integrate additional information about the syntax error such as error messages or error lines into the prompt. It is worth considering that incorporating additional information about syntax errors could potentially enhance the LLMC's performance to repair these syntax errors.

Additionally, we acknowledge that the greedy algorithm employed in our approach to minimize the number of test oracles might not be the most optimal solution for minimizing test oracles while maximizing MS. However, prior studies [23,204] using the same method to minimize the number of assertions have demonstrated its effectiveness in reducing the number of test oracles within test cases, along with its ease of implementation.

Construct Validity

We use the notions of killing mutants and bug detection as metrics to evaluate the effectiveness of test cases, given that the primary objective of testing is to reveal bugs. Coverage has been employed in various other studies to assess test case quality [22, 114].

On the other side, when the PUT undergoes changes, the corresponding test cases also need to be modified. The methodology behind *MuTAP* offers a distinct advantage in scenarios where the PUT has been altered by adding new functionality. In such cases, *MuTAP* can generate new test cases by creating mutants of the added sections of the PUT and then applying the augmentation steps. This advantage sets *MuTAP* apart from automatic test generation tools, which lack this capability. However, when it comes to scenarios where a portion of the PUT's functionality is removed or modified, potentially necessitating the removal of certain test cases, *MuTAP* shares the same limitation as automatic test generation tools. Detecting and removing test cases under these circumstances is a challenge. Nevertheless, in cases where unnecessary mutants are identified and removed, the oracle minimization step within *MuTAP* may prove effective in eliminating these redundant mutants.

It is important to note that the bugs present in mutants are artificial and might not directly correspond to real-world faults. To address this concern, we have employed the *Refactory* [69] dataset, a bug-repairing benchmark that contains real faulty programs developed by students.

External Validity

For our experiments, we used two datasets containing Python programming tasks, which could potentially pose external challenges to the validity of our findings. The requirement for executable Python programs is essential to run the generated tests against both the accurate and buggy versions (real or mutated) of *PUT* and this consideration guided our choice of datasets. However, since we didn't make any specific assumptions while selecting the dataset, our results can be extended to other Python programs.

Finally, it should be acknowledged that the technique proposed and the evaluations conducted in this study are conceptually adaptable to languages beyond Python. However, the current implementation of *MuTAP* is tailored for Python programs, meaning our existing results cannot be extended to cover other programming languages.

5.7 Chapter Summary

In this chapter, we introduced *MuTAP* as a means of enhancing and assessing the capacity of pre-trained LLMs to generate effective test cases. *MuTAP* initiates the process by instructing its LLMC to generate test cases through the use of *zero-shot* and *few-shot* learning techniques. After identifying and repairing any potential syntax errors and unintended behavior in the generated test cases, *MuTAP* evaluates their effectiveness via MT. Subsequently, it leverages any surviving mutants from the *PUT*, along with the initial inadequate test case to the initial prompt. By re-prompting the LLMC using this augmented prompt, *MuTAP* generates new test cases capable of detecting surviving mutants. The test cases generated by *MuTAP* outperform those generated by the state-of-the-art tool, Pynguin, as well as *zero-shot* and *few-shot* learning techniques, particularly in terms of bug detection. While the current version of *MuTAP* employs two distinct LLMs for test case generation in Python programs, its design and evaluation methodology are fundamentally adaptable to a variety of programming languages and models.

CHAPTER 6 REPRESENTING DOMAIN EXPERTISE OF DEVELOPERS IN AN EMBEDDING SPACE

6.1 Chapter Overview

As we discussed in the previous chapter, accurately assessing developers' expertise is crucial for selecting the right candidate to contribute to a project or fill a specific job role. One of the challenges in achieving this accurate assessment is how to effectively represent the domain expertise of developers. This challenge becomes even more pronounced when trying to obtain a comprehensive and precise representation of developers' domain expertise across different projects. While previous methods have achieved some success within a single software project, applying these methods to assess the domain expertise of developers based on their contributions across multiple projects has encountered certain limitations. In this chapter, we employ the *doc2vec* technique to represent the domain expertise of developers across multiple projects as embedding vectors and to assess their expertise from authored code fragments. To achieve this, we derived embedding vectors from various sources that provide evidence of developers' expertise. These sources include the descriptions of repositories they have contributed to, their issue resolving history, and the API calls made in their commits. We refer to this approach as *dev2vec* and demonstrate its effectiveness in representing and assessing the technical specialization of developers. Our proposed approach sheds light on the effectiveness of representing the technical expertise of developers using embedding vectors. It can serve as an initial filtering mechanism for recruiters and project managers.

6.2 Study Design

In this section, we first define the objectives of this chapter. Following that, we describe how we obtain ground truth labeling and provide details on how we extract data from software repositories on GitHub to compile our dataset. Figure 6.1 offers an overview of our approach for representing developers' domain expertise, which relies on three sources of their contributions.

6.2.1 Setting Objectives

The objective of this chapter is to address the problem of representing the domain expertise of developers based on their activities in various software projects on GitHub. The absence of an optimal representation of developers' expertise in previous studies, particularly in high-

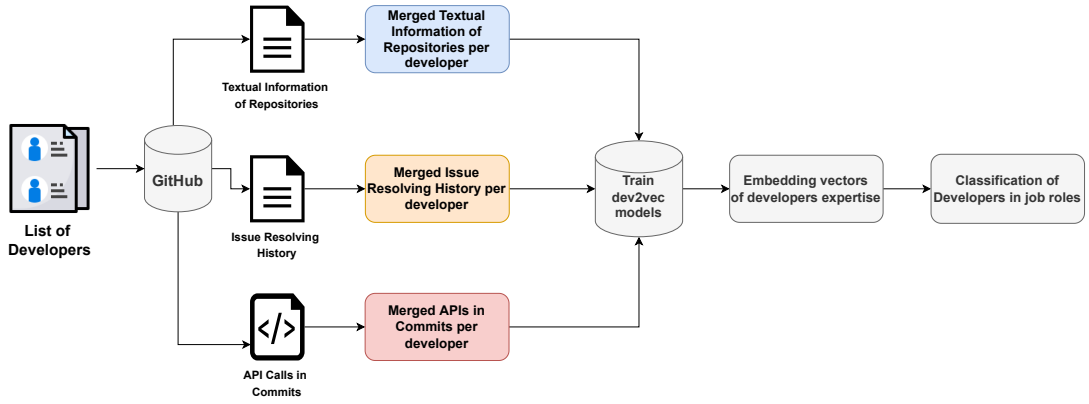


Figure 6.1 **Overview of the pipeline.** This is an overview of the proposed methods. Three separate sources of information are collected to represent developer expertise as embedding vectors.

dimensional data containing various developer activities across different software projects, can significantly impact the performance of related tasks, such as identifying proper candidates for contributing to a software project or a job role.

While previous efforts have demonstrated good performance in identifying experts within a single software project, they encounter challenges related to data sparsity and high dimensionality when applied to represent the domain expertise of developers across different types of activities and different software projects [30]. Additionally, these methods struggle to capture semantic or syntactic details [213].

Due to the popularity of `word2vec` and `doc2vec` and their proven effectiveness in similar tasks [164, 214], we hypothesize that these techniques can enhance the representation of developers' expertise within high-dimensional data.

`word2vec` [215] and `doc2vec` [33] are well-established methods for encoding words or aggregation of words into fixed-length vectors, resulting in lower-dimensional representations. These methods have been shown to improve the performance of various natural language processing tasks, such as information retrieval [216] and classification [217]. `word2vec` and `doc2vec` have demonstrated better performance compared to traditional approaches like bag-of-words [214]. Notably, `doc2vec` excels in this context due to its ability to represent developers' expertise in a lower-dimensional space that retains more semantic information about their domain expertise.

There are different types of information available concerning developers' activities on GitHub,

and these information sources can enhance the representation of developers' expertise. We gather data from three prominent sources of activities on GitHub to represent the domain expertise of developers [14, 145, 218]. We derive embedding vectors to represent developers' expertise from the metadata of repositories they contributed to, their issue resolving history, and the list of APIs used in the changes applied by developers to different source files. We have named these methods after their respective sources: *dev2vec:Repos*, *dev2vec:Issues* and *dev2vec:APIs*, respectively. Furthermore, we combine the output of these three methods by concatenating the embedding vectors from these three distinct information spaces, and we refer to it as *dev2vec:RIAs* (RIA stands for **R**epository, **I**ssue and **A**PI).

We assess the performance of each *dev2vec* model in reflecting the domain expertise of developers over a favored task in software engineering: identifying the technical job roles of developers. For this evaluation, we use a labeled dataset of GitHub developers who are categorized into five job roles: Backend, Frontend, Mobile, DevOps, and Data Scientist [30], as further elaborated in Section 6.2.2. Specifically, this chapter seeks to answer the following research question:

Can embedding techniques effectively capture the domain expertise of developers from their contributions to various software projects?

To address this research question, this chapter has made the following contributions:

- Propose a method for representing the domain expertise of developers using embedding vectors, achieved by applying `doc2vec` to various sources of information: *dev2vec:Repos*, *dev2vec:Issues*, and *dev2vec:APIs*.
- Propose the aggregation of three distinct types of information about developers' activities across various projects on GitHub. This approach involves representing the domain expertise of developers by concatenating the embedding vectors derived from different information sources: *dev2vec:RIAs*.
- Evaluate the effectiveness of the *dev2vec* method in representing the domain expertise of developers within the context of a prevalent problem in the software engineering domain: assessing the technical specialization of developers.

6.2.2 Data Collection

To determine the domain expertise of developers, we rely on three sources of information available on GitHub. The first source is the description of repositories to which they have

contributed. The second source concerns their history of resolving issues across various repositories. The final source entails the list of APIs used in the modifications (commits) applied by developers in different repositories. An overview of the data collection process is depicted in Figure 6.1. In the subsequent section, we first outline our approach to obtain ground truth labels. Following this, we explain the methods used to collect data for these three categories.

Ground Truth

Our training and validation method relies on the availability of labeled data that accurately represents developer expertise, referred to as the ground truth. To achieve this, we used a labeled dataset of developers as described in [30], comprising 1662 individuals who were categorized into five distinct job roles:*Backend, Frontend, Mobile, DevOps, and Data Scientist*. Notably, the expertise labels were not generated using GitHub's data; instead, the labels were derived from StackOverflow data, associating users with GitHub profiles. This was achieved through the application of various regular expressions to the users' profile information on StackOverflow. Additionally, the GitHub usernames of these developers were collected to establish a connection with their GitHub pages.

It's worth noting that 20% of these developers had multiple labels assigned to them. However, for the purposes of our study, we focused only on developers with a single domain of expertise, resulting in a final selection of 1340 developers with single job roles. From this group, we were able to successfully locate the GitHub profiles of 1272 developers along with their corresponding usernames. The remaining 68 developers might have either changed their usernames or closed their GitHub accounts. Figure 6.2 represents the distribution of developers across these five job categories in this dataset.

Textual Information of Repositories

Drawing upon insights from prior studies, information about the repositories that developers have contributed in the past serves as a valuable reference for assessing their skills and domain of expertise [145, 218]. In GitHub, each repository has various features in natural language, which effectively represent the repository's domain. These features encompass *Name, Tags, Topics, and ReadMe*. "Name" denotes the repository's title and is a mandatory field. Repository names are often selected to convey the project's objectives to some extent, but they are concise and not very descriptive (e.g. "eth-tester-rpc [1]").

[1] `https://github.com/voith/eth-tester-rpc`

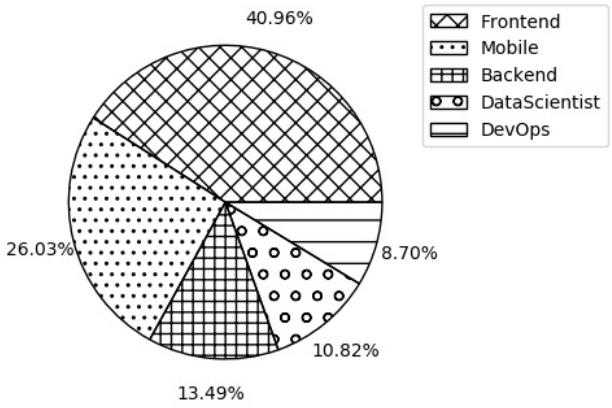


Figure 6.2 **The distribution of developers in different job roles.** The distribution of 1272 developers in five job roles: Backend, Frontend, Mobile, DevOps, and Data Scientist. This distribution is imbalanced.

"Tags" can be assigned to a repository to define its domain, programming language, or technologies employed within the project. Tags, however, are not mandatory and may be left unassigned (i.e. #ethereum, #python, #crypto). The "Topic" or "About" section consists of a brief description, typically limited to two or three sentences, presenting the repository's goals. The "ReadMe" file provides in-depth information, encompassing the project's applications, the functionality of individual source files, the programming languages used within the repository, and more. Both "Topic" and "ReadMe" are optional components.

We combine the content of all these four features for each repository, referring to it as the textual information of repositories. Subsequently, we gather the textual information from all repositories and combine it on a per-developer basis. This process yields a document for each developer, containing the textual information derived from repositories to which they have contributed. If a repository is forked, we identify the parent repository and, if the developer has made any contributions (commits) to the parent repository, we also collect its description. However, if a developer's commits on a repository amount to fewer than 5 lines in total (comprising both added and removed lines), we exclude that repository from the developer's contributions. In total, we aggregate the textual information from a substantial 58K repositories spanning 1,272 developers.

Issue Resolving History

The issue resolving history of developers across various projects on GitHub is another valuable source containing relevant information regarding their domain expertise, as evidenced in prior research [14, 129]. Developers can contribute to issues in GitHub repositories in

diverse ways. An issue can be assigned to a single developer or a list of developers, and developers can also create issues within a repository. Also, developers often identify issues within a repository while contributing to it or while using it for their individual purposes, and they subsequently acquire the knowledge to address these issues based on the solutions. Alternatively, developers may participate in issue discussions because they possess expertise in resolving the issue or because it related to their domain of knowledge. Issue reports are recognized as a significant avenue for sharing knowledge during the development process and contain valuable information about a developer's skills and expertise [160].

We collectively regard these three types of activities as contributions by developers to solving an issue on GitHub, as they signify that developers possess the sufficient expertise to participate in of its phases, from creation to resolution. We refer to this as the Issue Resolving History of developers.

For our study, we have collected the headers and bodies of 60K issues assigned to 1,272 developers. The content from both the header and body of each issue is combined to form a comprehensive record. Subsequently, for each developer, we aggregate the content from all their issue resolving history. Akin to the textual information extracted from repositories, the outcome of this stage is a document for each developer, with each document containing the issue resolving history of that developer.

API Calls per Commits

The main contributions of developers on different repositories are in the format of commits. Another valuable source of information, often considered as evidence of developers' expertise, is the list of API calls found in commits across different GitHub repositories, as discussed in previous studies [28, 139].

Developers alter one or more source files by submitting a commit, and each source file typically contains a list of APIs that developers may either modify or leave unchanged. Previous research has shown that when developers modify a source file, they typically possess a basic familiarity with the APIs within that file, more so than with random APIs that have no connection to their contributions [164, 219]. As a result, we assume that developers have a fundamental understanding of the APIs used in the source files they modify. However, it's important to note that this assumption may introduce some degree of noise into the collected APIs. To mitigate the impact of this noise, we focus on APIs that appear in developers' contributions at least five times.

To accomplish this, for each commit, we extract all language-specific source files associated

with that commit after its submission. Similar to the process outlined in Section 6.2.2, if the total size of a developer's commit (i.e., the combined number of added lines and removed lines) on a source file is less than five lines, we exclude the list of API calls from that commit in the developer's contribution. In total, we extract source files from approximately 21 million commits, which represent the modified versions of the source files following each commit. Consequently, for each developer, we list all the APIs found in the source files of their commits. For a more detailed explanation of the API collection process from developers' commits, please refer to Section 6.3.3.

6.3 Approach

This section outlines how developers' expertise can be represented using embedding vectors through the three proposed methods: dev2vec:Repos, dev2vec:Issues, and dev2vec:APIs. It also describes the process of training various models using the data sources discussed in Section 6.2.2. Furthermore, we provide an explanation of the method known as dev2vec:RIAs, which involves aggregating these three embedding vectors of expertise.

6.3.1 The embedding of domain expertise with *doc2vec*

Evaluating the expertise of developers across various software projects and programming languages, regardless of the data source, leads to an increase in the number of unique tokens. For instance, after cleaning the repository descriptions, we are left with 18 million distinct tokens. If we include API calls, this number escalates to 100 million tokens [164]. While previous methods, such as the bag-of-words approach, demonstrate good performance in identifying experts within a single software project, they do not scale well for representing developers' expertise across diverse projects.

As we have discussed, one solution to tackle these issues is to encode this information into fixed-length vectors with considerably lower dimensions, using algorithms such as `word2vec` and `doc2vec`. `Word2vec` [215] learns a numerical vector representation for each word within a corpus of documents using two different algorithms, CBOW and skip-gram. Typically, the vector size is much smaller than the vocabulary dimension, typically ranging from 100 to 300 [215].

An extension of `word2vec` is `doc2vec`, which represents each document as an embedding vector and is trained by predicting the words within a document in a given corpus. The input text provided to `doc2vec` can have variable lengths, spanning from a single sentence to a lengthy document [33]. In `doc2vec`, each document is assigned a unique paragraph

identifier or tag. This tag can consist of a single identifier or a list of identifiers for each document. `Doc2vec` learns an embedding vector for each document and each word within the documents. This embedding vector effectively represents the content of the document.

The `Doc2vec` algorithm consists of two stages. The first stage is the training phase, which offers two approaches for training document vectors and word vectors. The first approach is the Distributed Memory model (DM). In this approach, the document embedding vector functions as an additional input word. The average (or concatenation) of this vector with the embedding vectors of the other input words is used to predict the output word. The embedding vectors of documents serve as a form of memory representing the document's content. The second approach is the Distributed Bag Of Words (DBOW), where the document embedding vector is trained to predict randomly selected words in the output [33].

The second stage is the inference phase (or prediction phase). During this phase, the vectors of new documents are inferred by concatenating their word vectors, and the process consists of several iterations until convergence. Words that were not seen during the training phase are disregarded.

6.3.2 dev2vec:Repos and dev2vec:Issues

The dev2vec:Repos model employs the data gathered in Section 6.2.2 to train a `doc2vec` model. Each developer's textual information from repositories is treated as a document. After tokenization and cleaning, we create tagged-document data by assigning each document a tag based on the developer's username or Author ID. The first pipeline in Figure 6.3 illustrates the data entries used to construct *dev2vec:Repos*. It includes *[developer id]* as the tag or paragraph ID and *[w1,w2,...,wn]* as the document's content. *[w1,w2,...,wn]* represents words extracted from the descriptions of repositories to which a developer has contributed in the past.

We divide these documents into a training set and a test set. Section 6.4.1 provides further details on the experimental setup and the data partitioning process. We train the `doc2vec` model on tagged documents from developers in the training set and infer vectors for developers in the test set based on the content of their documents. It's worth noting that during the inference stage, if a new word is encountered in the textual information of repositories for a developer in the test set (a word that was not seen in the training set), it is ignored.

For the dev2vec:Issues model, we follow a similar procedure to dev2vec:Repos, with the distinction that we use the data described in Section 6.2.2. In this case, words are gathered from the titles and bodies of issues that a developer has resolved, created, or contributed to

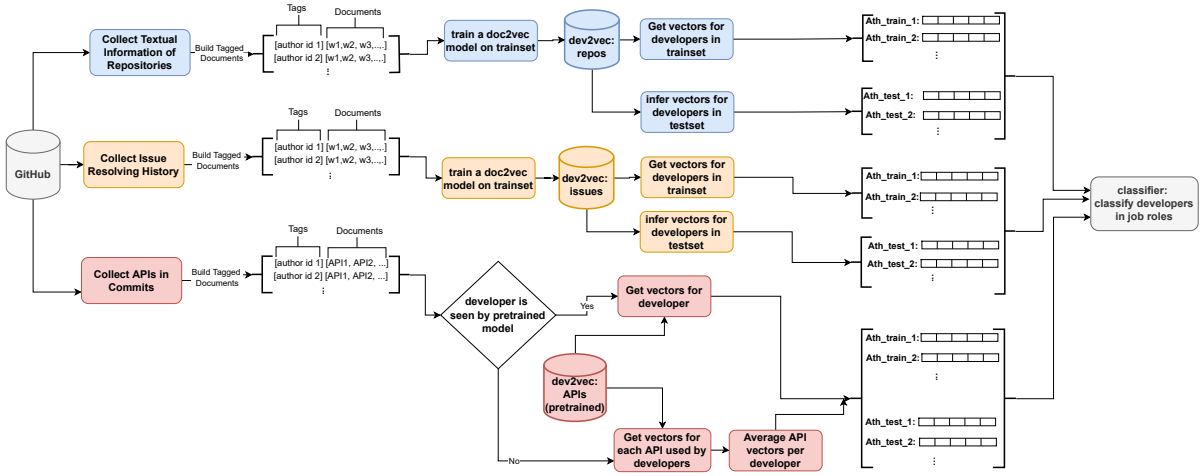


Figure 6.3 **Detailed view of the pipeline.** A more detailed view of the three proposed methods, dev2vec:Repos, dev2vec:Issues and dev2vec:APIs. The embedding vectors generated by each of these models represent the expertise of developers in different embedding spaces.

in the past. The format of the data used to train *dev2vec:Issues* is presented in the second pipeline in Figure 6.3. Similar to the inference stage of dev2vec:Repos, any new word that surfaces in a developer's issue resolving history in the test set is disregarded. Additional information on the experimental setup for these two models can be found in Section 6.4.1.

6.3.3 dev2vec:APIs

The Dev2vec:APIs model employs a pre-trained `doc2vec` model [164], which has been trained on API calls within source files collected from GitHub. We selected this model due to its extensive training dataset, comprising 36K projects, 690K authors, and 1.2 billion source files collected after submitting commits. Training such a large-scale model is time-consuming and requires substantial computational resources. Moreover, we cannot infer the embedding vector for new developers using this model directly (not seen in their training phase) due to the granularity level of data entries in their model. To address this limitation, we introduce a novel approach and named it Dev2vec:APIs to predict developers' expertise based on their API usage. This enables us to indirectly compare this model with Dev2vec:Repos and Dev2vec:Issues. In the subsequent section, we will provide a more in-depth exploration of the various steps of Dev2vec:APIs.

Parsing source files

Since we use the pre-trained model in [164], we follow the same structure to collect the API calls in the source files. They collected the list of libraries (imports) in the version of source files after submitting a commit for 17 programming languages: C, C#, Java, FORTRAN, Go, JavaScript, Python, R, Rust, Scala, Perl, Ruby, Dart, Kotlin, TypeScript, Julia and Jupyter Notebook (iPython). We parse each source file based on the syntax of its programming language and collect the list of its libraries for each developer. We match the extension of each source file after submitting a commit with its programming language. Then, based on the grammar of its programming language, we apply the related regular expression to collect its list of libraries. For example, for Python, we use this regular expression to find libraries: "$(? : from|import)\s + \w * .+; *$".

Obtaining embedding vectors of developers expertise

In the pre-trained model [164] the granularity level of the input data is fed into the `doc2vec` model, is per commit (changed version of source file after submitting a commit). The model used a list of identifiers: *[programming language, repository, timestamp, author id]* as the document tag and the list of APIs in the changed version of the source file as the content of the document or words. The Author id belongs to the developer who modified this source file by submitting the commit.

In the inference stage of `doc2vec`, it is necessary to infer document vectors for documents at the same granularity level as those in the training set. However, we can obtain or retrieve the vectors of documents and words that were encountered and trained during the training phase. Out of the 1272 developers in our dataset, only 40 of them are present in their dataset. We directly extract the embedding vectors for these 40 developers from their trained model.

One potential solution for predicting the embedding vectors for the remaining developers in our dataset, using this pre-trained model, is to leverage the embedding vectors of APIs from this model. This approach involves concatenating the vectors of all the APIs found in each developer's contributions. Similar to the inference stage of `doc2vec`, we omit the API calls that were not encountered by this pre-trained model. Subsequently, we calculate the average of all these API embedding vectors to create a single vector that represents the developer's domain expertise. The lower pipeline in Figure 6.3 outlines the steps of 'dev2vec:APIs'.

6.3.4 Dev2vec:RIAs

The previous sections generated embedding vectors from single sources. We now turn to a method to aggregate sources and summarize different types of developers' activities in a single embedding vector. Two of the data sources are in natural language and one is in the programming language. Since embedding vectors are trained in three different embedding spaces, we cannot aggregate them with simple mathematical functions such as average, max, or min. To address this shortcoming, we evaluate the efficiency of concatenating the embedding vectors of developers' expertise generated from these three spaces. Figure 6.4 shows the pipeline for this section.

For more details on how concatenation performs here, suppose that the embedding vectors for dev_i generated by *dev2vec:Repos*, *dev2vec:Issues* and *dev2vec:APIs* are $v_{(dev_i,repo)} \in \mathbb{R}^n$, $v_{(dev_i,issue)} \in \mathbb{R}^m$ and, $v_{(dev_i,api)} \in \mathbb{R}^p$ respectively. Since embedding vectors of these three models are obtained from different sources we did not control the size. In Section 6.4.1, we return to the size of embedding vectors.

The set of embedding vectors for each developer dev_i are concatenated by mapping the space into $C : \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^p \rightarrow \mathbb{R}^{n+m+p}$ according to Equation 6.1. Where we use $+\!\!+$ as the concatenation operator for vectors.

$$C(X,Y,Z) = \{X +\!\!+ Y +\!\!+ Z | X \in \mathbb{R}^n, Y \in \mathbb{R}^m, Z \in \mathbb{R}^p\} \tag{6.1}$$

We opted for this concatenation technique because aggregating (i.e., averaging) embedding vectors obtained from three distinct sources and three different embedding spaces was not feasible. While there may be some redundancy among the features within concatenated vectors, in general, it is left to the classifier to determine the most relevant features. However, as part of an additional investigation, we explored whether there were any linear correlations among the features within the concatenated vectors and whether their combination could enhance the classifier's performance.

To undertake this investigation, we employ a linear combination of features, selecting the resulting transposed features that offer greater interpretability while discarding the less significant ones. For this purpose, we utilize Principal Component Analysis (PCA), a feature extraction technique. PCA identifies a linear combination of input features designed to maximize the variance captured in the transformed components. It effectively removes the least essential features while preserving the valuable ones [220]. Previous studies have employed PCA to project embedding vectors into lower-dimensional vectors for various purposes, including optimizing the feature vectors [221] and visualizing them [222].

We use both concatenated vectors before and after applying PCA and then compare their performance in representing developers' expertise. This evaluation aims to determine whether there exist relevant linear correlations among the features in the concatenated vectors.

6.4 Evaluation

In this section, we will begin by outlining the evaluation of each dev2vec model's efficiency in terms of developers' technical specialization. Following that, we will introduce a state-of-the-art method for comparison, detailing the setup for each dev2vec model. To conclude, we will address our research questions in the context of the results obtained.

6.4.1 Experimental Setup

To assess the effectiveness of each dev2vec model in capturing developers' expertise, we employ a common software engineering task: classifying developers into distinct job roles. We consider five roles, namely *Backend, Frontend, Mobile, DevOps, and Data Scientist*. As depicted in Figures 6.3 and 6.4, we input embedding vectors obtained from each model into classifiers to acquire insights about the developer roles.

We utilize three widely recognized classifiers: SVM, Random Forest, and Logistic Regression. The performance of these models is compared by evaluating key metrics such as *Precision, Recall*, and *F1-score*.

Comparable Method

Numerous studies have employed the bag-of-words technique to represent developers' expertise [14,145]. In our work, we compare our proposed methods with state-of-the-art techniques that focus specifically on developers' domain expertise across different GitHub projects. We replicate a recent study [30] that centers on representing developers' expertise using GitHub information and addresses the classification problem of developers within their job roles. Our study utilizes the same set of developers as [30], who are categorized into five distinct job roles: *Backend, Frontend, Mobile, DevOps, and Data Scientist*.

In their study, [30] utilized GitHub data on developer contributions across various projects to capture their expertise. They gathered developer biographies, repository names, programming languages used in repositories, repository topics, and repository dependencies (libraries) across diverse projects as a collective set of information reflecting developers' domain expertise. They applied the bag-of-words technique to construct a vector for each developer based

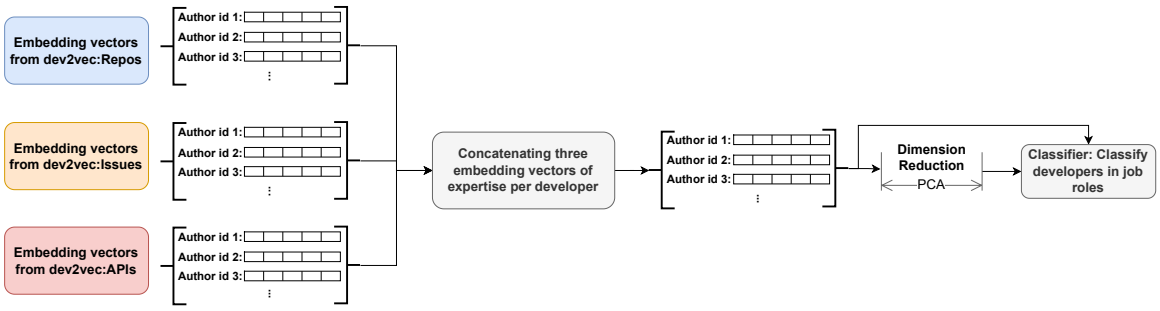


Figure 6.4 **Dev2vec:RIAs.** We concatenate three embedding vectors generated in three different spaces of *dev2vec:Repos*, *dev2vec:Issues* and *dev2vec:APIs* to represent the expertise of developers

on the frequency of each feature within their contributions. Given the high dimensionality of the collected data, they employed feature selection through a correlation technique, removing features with significant correlations within each category (repository names, topics, languages, and dependencies). This process resulted in 1471 features out of nearly 18,000. We referred to this method as "SOA:bow".

dev2vec Setup

`Doc2vec` has several hyper-parameters. We empirically optimize important parameters and keep the default values for the rest. The first parameter is the vector size or embedding size. The words and documents embedding vector share the same size. Another parameter is window size which defines the number of words considered around a target word to learn its embedding. Another important parameter is the minimum frequency for a word to be considered (min_count). Another parameter is the negative sampling size. The window size around a word considers positive sampling words for the target word in `doc2vec`. The words out of this window are considered as negative labels or uncorrelated words to the target words. Since the size of vocabularies is enormous, the negative sampling technique randomly downsampled these words into negative sampling size to speed up the convergence. Finally, we should find the number of epochs to repeat the training and the inference stages. For *dev2vec:APIs* model, we use the pre-trained model in [164]. This model uses DBOW to do not attempt to the order of APIs because the order of APIs in a source file is not important. Table 6.1 shows a summary of the experimental setup.

For cross-validation, we divide our developers into different sets: 80% of the trainset, 10%

Table 6.1 **Experimental Setup for dev2vec models.** The adopted range to identify the best value is shown under each parameter.
The range $[a, b] \in N$.

Model	vector size [50,300]	window size [2,50]	min_count [1,10]	algorithm (DM or DBOW)	negative sampling (default)	epochs [5,30]
dev2vec:Repos	230	5	5	DM	5	15
dev2vec:Issues	150	5	5	DM	5	20
dev2vec:APIs	200	30	5	DBOW	20	10

of the validation set, and 10% of the testset. Developers in testset are not in trainset or validation set. To cross-validate the result, for *dev2vec:Repos* and *dev2vec:Issues*, we search for the best parameter through cross validation over the validation set. After learning the vector representations for the developers in trainset, we feed them to a classifier to learn a predictor of the developers' job roles. For the inference stage, we infer vectors for developers in testset through these two dev2vec models and use them as separate testsets of the classifiers.

For *dev2vec:API*, since there is no inference stage, we build the embedding vectors of the developers by averaging the embedding vector of APIs in the list of their API expertise. We use the embedding vectors of developers in trainset to train the classifier and the embedding vector of developers in testset to test it. We apply the same process for the comparable method, "SOA:bow".

6.4.2 Experimental Result

In this section, we address our main research question by breaking it into three different sub-research questions:

RQ1: How effective are embedding vectors to represent the domain expertise of developers across various software projects compared to the state-of-the-art methods?
RQ2: How sensitive is the performance of dev2vec to the source of information (repositories, issues, and API)?
RQ3: How effective is the concatenation of expertise embedding vectors from the different information sources?

In the following of this section, we discuss our findings to address each research question, separately.

RQ1: How effective are embedding vectors to represent the domain expertise of developers across various software projects compared to the state-of-the-art methods?

To answer RQ1, we consider the method described in Section 6.4.1, as a state-of-the-art method and call it "SOA:bow". We apply all methods to the same dataset. Since the distribution of developers in different job roles is imbalanced and 41% of developers are *"frontend"*, we consider a baseline as a majority-rule classifier that predicts all developers as *"frontend"*. For the state-of-the-art, SOA:bow, and all dev2vecs, we apply Random Forest to classify developers in different job roles.

Since the classes in our dataset are imbalanced, we apply the Macro-Weighted Precision, Recall, and F1-score to report the performance. With Macro-Weighted metrics, the proportion of each class is weighted by its relative number of examples available in the dataset.

Table 6.2 The performance of a classifier across five job roles based on three dev2vec methods, *dev2vec:Repos*, *dev2vec:Issues* and *dev2vec:APIs*, compared to a state-of-the-art, *SOA:bow*, and a baseline. Since the classes are imbalanced, Macro-Weighted Precision, Recall, and F1-score are used to report the performance.

Methods	Macro-Weighted Precision%	Macro-Weighted Recall%	Macro-Weighted F1-score %
Baseline	18.73	43.28	26.15
SOA:bow	41.73	43.93	42.35
dev2vec:Repos	60.78	61.90	**60.02**
dev2vec:Issues	64.75	65.04	**63.08**
dev2vec:APIs	56.90	55.10	**55.51**

We can find in Table 6.2 that all dev2vec methods and the state-of-the-art, SOA:bow, show a better performance than the baseline. The Macro-Weighted Precision to classify developers in their job roles with SOA:bow is 41.73%, while it is equal to 18.73% for the baseline. This precision is equal to 60.78%, 64.75% and 56.90% for *dev2vec:Repos*, *dev2vec:Issues* and *dev2vec:APIs*, respectively.

dev2vec:Issues shows 64.75% Macro-Weighted Precision for classifying developers in their job roles. It improves the precision of the classification task up to 23.02% compared to the SOA:bow. The Macro-Weighted F1-score for *dev2vec:Repos*, *dev2vec:Issues* and *dev2vec:APIs* equals 60.02%, 63.08%, 55.51% respectively compare to the SOA:bow with 42.35% F1-score.

Finding 1: Representing the domain expertise of developers across different software projects in embedding vectors shows better performance than both a baseline and a recent state-of-the-art in classifying developers in their technical job roles. The proposed dev2vec methods improve F1-Score at least 13.16% with *dev2vec:APIs* and at most 20.73% with *dev2vec:Issues*.

RQ2: How sensitive is the performance of dev2vec to the source of information (repositories, issues, and API)?

To answer RQ2, we compare three different dev2vec models, *dev2vec:Repos*, *dev2vec:Issues*, and *dev2vec:APIs*, that are trained on three different sources of activities to investigate their impact on representing the expertise of developers in embedding vectors.

Table 6.3 shows the result of all three dev2vec methods for classifying developers with three types of classifiers. We apply SVM [223], Random Forest [224] and Logistic Regression [225]. The metrics are measured in each job role to study the difference in performance between different classes separately. The table also presents the results of SOA:bow on each job role.

Among the three dev2vec models, *dev2vec:Issue* shows better performance than *dev2vec:Repos* and *dev2vc:APIs* in classifying developers in different job roles. For example, the precision of *dev2vec:Issue* in classifying developers with *Data Scientist* role is 80% and the recall for *Frontend* developers is 90.48%. It also improves the F1-score for *DevOps* developers up to 26.82% with Logistic Regression classifier. The *dev2vec:repos* shows better performance than *dev2vc:APIs*. For example, *dev2vec:APIs* has 46.43% and 36.11% precision and recall respectively for *Mobile* developers with SVM. However, it improves to 53.82% and 46.42% with *dev2vec:Repos*.

In addition to the above comparison, all three dev2vec models improve the performance of the classification task in all job roles compared to the state-of-the-art, *SOA:bow*. For example, in *SOA:bow* and SVM classifier, recall for *Backend* and *DevOps* developers are 20% and 23.08%, respectively. It increases to 31.25% and 44.45% with dev2vec models. As another example, the precision for *Data Scientist* class in bag-of-words with Random Forest classifier is 46.15% and it increases into 78.57%, 80% and 64.71% with *dev2vec:Repos*, *dev2vec:Issues*, and *dev2vec:APIs*, respectively.

Some job roles show better performance. For example, “Data Scientist” developers are more accurately classified compared to “Backend” developers. This is due to the intersection between the activities of developers in different roles. We discuss it more in Section 6.6.

Table 6.3 The result of three different classifiers on the embedding vector representation of developers' expertise in three different spaces. The "Pre", "Rec", and "F1" are referring to Precision, Recall, and F1-score, respectively.

	SVM											
	SOA:bow			**Dev2vec:Repos**			**Dev2vec: Issues**			**Dev2vec: APIs**		
Job Role	**Prec%**	**Rec%**	**F1%**	**Prec%**	**Rec%**	**F1%**	**Prec%**	**Rec%**	**F1%**	**Prec%**	**Rec%**	**F1%**
Frontend	53.52	73.08	61.79	63.28	85.40	72.69	**66.67**	**90.48**	**76.77**	61.11	78.57	68.75
Backend	33.33	20.00	25.00	37.50	28.64	31.79	**55.56**	**31.25**	**40.00**	31.25	27.78	29.41
Mobile	32.26	30.30	31.25	53.82	46.42	48.17	**59.09**	**52.00**	**55.32**	46.43	36.11	40.63
DataScientist	60.00	42.86	50.00	76.04	61.76	67.82	**80.00**	**72.73**	**76.19**	66.67	66.67	66.67
DevOps	37.50	23.08	28.57	68.89	37.17	48.08	**80.00**	**44.45**	**57.14**	37.50	30.00	33.33
	Random Forest											
	SOA:bow			**Dev2vec:Repos**			**Dev2vec: Issues**			**Dev2vec: APIs**		
Job Role	**Prec%**	**Rec%**	**F1%**	**Prec%**	**Rec%**	**F1%**	**Prec%**	**Rec%**	**F1%**	**Prec%**	**Rec%**	**F1%**
Frontend	55.56	68.18	61.22	64.29	84.91	73.17	**67.27**	**88.10**	**76.29**	56.00	73.68	63.64
Backend	31.25	20.00	24.39	50.00	27.27	35.29	**55.56**	**31.25**	**40.00**	33.33	31.25	32.26
Mobile	38.46	40.54	39.47	53.13	45.95	49.28	**56.00**	**56.00**	**56.00**	50.00	41.03	45.07
DataScientist	46.15	46.15	46.15	78.57	64.71	70.97	**80.00**	**72.73**	**76.19**	64.71	64.71	64.71
DevOps	20.00	15.38	17.39	50.00	25.00	33.33	**75.00**	**33.33**	**46.15**	30.00	21.43	25.00
	Logistic Regression											
	SOA:bow			**Dev2vec:Repos**			**Dev2vec: Issues**			**Dev2vec: RAIs**		
Job Role	**Prec%**	**Rec%**	**F1%**	**Prec%**	**Rec%**	**F1%**	**Prec%**	**Rec%**	**F1%**	**Prec%**	**Rec%**	**F1%**
Frontend	46.43	47.27	46.85	74.47	68.63	71.43	**76.92**	**71.43**	**74.07**	60.53	50.00	54.76
Backend	23.53	19.05	21.05	25.00	30.77	27.59	**50.00**	**31.25**	**38.46**	20.00	28.57	23.53
Mobile	37.50	40.00	38.71	50.00	43.24	46.38	**53.13**	**68.00**	**59.65**	42.86	42.86	42.86
DataScientist	40.00	46.15	42.86	62.50	58.82	60.61	**66.67**	**72.73**	**69.57**	50.00	52.63	51.28
DevOps	25.00	23.08	24.00	26.67	50.00	34.78	**50.00**	**55.56**	**52.63**	25.00	26.67	25.81

Finding 2: The performance of dev2vec model is sensitive to embedding vectors learned from different sources of developers' activities. The *dev2vec:Issues* shows a better performance in representing the expertise of developers in embedding vectors. The *dev2vec:Repos* and *dev2vec:APIs* are in second and third place, respectively.

RQ3: How effective is the concatenation of expertise embedding vectors from the different information sources?

We concatenate three embedding vectors from three different spaces of their activities, as explained in Section 6.3.4. The size of the final embedding vector that represents the expertise of developers is 580.

We feed these vectors into a Random Forest classifier to predict developer job roles. We also apply PCA on these embedding vectors to investigate if there is any linear correlation between the features in concatenated vectors. We reduce the dimension from 580 to 50, 100, 200, 250, and 300, and each time, train a new classifier for predicting the developers' job roles. Table 6.4 shows the results for embedding vectors with and without applying PCA. Based on this table, concatenating embedding vectors from different spaces improves the performance of the classifier only in two job roles, "Frontend" and "Data Scientist". But for the rest of the job roles, the performance is as good as *dev2vec:Issues*.

Applying PCA to reduce dimension by linearly combining features has an adverse effect on the performance of the classifier. Increasing the number of dimensions improves performance. However, even with "PCA-300", the performance of the classifier is not as good as *dev2vec:Issues*.

The adverse impact of PCA on results in Table 6.4 shows that there are no linearly correlated features in the concatenated vectors. We infer that because PCA is a linear technique, it has an opposing or no significant effect when it is applied to features that are not linearly correlated. In other words, applying PCA on embedding vectors to reduce dimensionality by combining non-linear features may have the adverse effect of mapping embedding features in a space where they are no longer effective.

Finding 3: Concatenating embedding vectors from different spaces to represent the expertise of developers, improves the performance of the classifier for different cases, and there are no relevant features in concatenated embedding vectors that are linearly correlated.

Table 6.4 The performance of *dev2vec:RIAs* with different dimensionality reduction levels in classifying developers in their job roles

Reduction	Roles	Prec%	Rec%	F1%
None	**Frontend**	69.09	88.37	**77.55**
	Backend	55.56	31.25	40.00
	Mobile	56.00	56.00	56.00
	DataScientist	81.82	81.82	**81.82**
	DevOps	75.00	33.33	46.15
PCA-50	**Frontend**	55.36	73.81	63.27
	Backend	30.77	23.53	26.67
	Mobile	40.00	32.00	35.56
	DataScientist	58.33	58.33	58.33
	DevOps	40.00	20.00	26.67
PCA-100	**Frontend**	56.14	76.19	64.65
	Backend	30.77	23.53	26.67
	Mobile	42.11	32.00	36.36
	DataScientist	58.33	58.33	58.33
	DevOps	40.00	20.00	26.67
PCA-200	**Frontend**	60.71	80.95	69.39
	Backend	38.46	29.41	33.33
	Mobile	47.37	36.00	40.91
	DataScientist	58.33	58.33	58.33
	DevOps	50.00	30.00	37.50
PCA-250	**Frontend**	61.82	80.95	70.10
	Backend	38.46	29.41	33.33
	Mobile	47.37	36.00	40.91
	DataScientist	61.54	66.67	64.00
	DevOps	50.00	30.00	37.50
PCA-300	**Frontend**	62.50	83.33	71.43
	Backend	41.67	29.41	34.48
	Mobile	50.00	40.00	44.44
	DataScientist	61.54	66.67	64.00
	DevOps	60.00	30.00	40.00

6.5 Discussion

The results presented in this chapter shed light on addressing the problem of representing the domain expertise of developers across various software projects and different types of activities. Since different types of activities on GitHub originate from distinct sources of information, previous studies have encountered limitations in adequately considering the diverse range of developer activities to represent their expertise. Our findings indicate that techniques like doc2vec have the potential to represent developers' domain expertise across different software projects through the use of embedding vectors. Notably, our method outperforms both a baseline model and a recent state-of-the-art approach in classifying developers in their respective technical job roles.

Our proposed *dev2vec* methods demonstrate significant improvements in F1-Score, with *dev2vec:APIs* showing a minimum improvement of 13.16%, and *dev2vec:Issues* yielding the highest improvement at 20.73%. It is worth noting that the performance of the *dev2vec* model varies depending on the source of the embedding vectors learned from different types of activities. *dev2vec:Issues* excels in representing developers' expertise in embedding vectors, with *dev2vec:Repos* and *dev2vec:APIs* as second and third place, respectively.

Furthermore, the concatenation of embedding vectors from distinct information spaces to represent developers' expertise proves to enhance the classifier's performance in various scenarios in predicting developers' job roles.

However, there are some limitations to consider, particularly with *dev2vec:APIs*. Due to the inability to directly use the pre-trained model to predict embedding vectors of expertise for new developers, we employ an approach that averages the embedding vectors of APIs associated with a developer's activities. This method may impact the effectiveness of these embedding vectors in capturing the expertise of developers. This is one of the reasons why *dev2vec:APIs* exhibits lower performance compared to the other two *dev2vec*s models.

Additionally, our results reveal variations in performance across different job roles. The classifiers demonstrate better performance in job roles such as Frontend and Data Scientist compared to Backend or Mobile for both the *SOA:bow* and all *dev2vec* models. In practical terms, we recognize that there are overlaps in developers' skill sets in different job roles. For instance, there are shared skills and project involvement between Frontend and Backend developers, as well as Mobile developers. However, these intersections are less pronounced in the case of Data Scientists when compared to Frontend or Mobile categories.

To illustrate the correlation between different job roles, we examine the similarity between words within and across topics using the *Cosine Inter_Intra Topics* method [226]. We apply

this method to the state-of-the-art *SOA:bow* dataset, which we have adopted as our ground truth.

In Figure 6.5, rows represent the job roles, and columns represent the centroids of each role. We observe that the diagonal values for Frontend and Data Scientist roles exceed the off-diagonal values. This indicates that the similarity among developers within these two job roles is greater than the similarity between them and developers in other roles. However, for Backend and Mobile developers, the diagonal and off-diagonal values are very close. Even the similarity between Mobile and Frontend developers surpasses the similarity among Mobile developers themselves. This observation could potentially explain the lower performance of classifiers in some job roles like Backend and Mobile.

Our approach at full length is a combination of *doc2vec* and machine learning classifiers. We apply *doc2vec* to developers' activities and learn the embedding vector representation of their expertise. These trained models are then applied to predict embedding vectors for new developers in the test set. In the final step, we assess the performance of these vectors in representing developers' expertise using different classifiers, such as Random Forest in a scenario of predicting their job roles.

It's important to note that our dataset includes 1272 developers categorized into five classes, which may not be sufficient to train an end-to-end supervised neural network for the classification task. Based on our observation of previous studies on multi-label text classification neural networks, a minimum training set of 10,000 documents is needed to achieve good classifier performance [227–229].

As we also discussed in the previous chapter, assessing the expertise of developers encompasses various aspects, including both their soft skills, such as teamwork and communication, and technical skills, such as programming languages, algorithm design, and libraries. This study primarily focuses on representing the domain expertise of developers across their contributions to different GitHub projects, serving as an initial filtering tool for recruiters and project managers. Further evaluations and interviews are necessary for selecting the most suitable candidate.

Moreover, the benefits of embedding vectors of developers' expertise extend beyond classifying job roles. These vectors can be applied in various ways in software engineering, including identifying suitable candidates for job posts or new project contributors, as well as matching developers with similar expertise in a given domain. These potential applications warrant further exploration in future research.

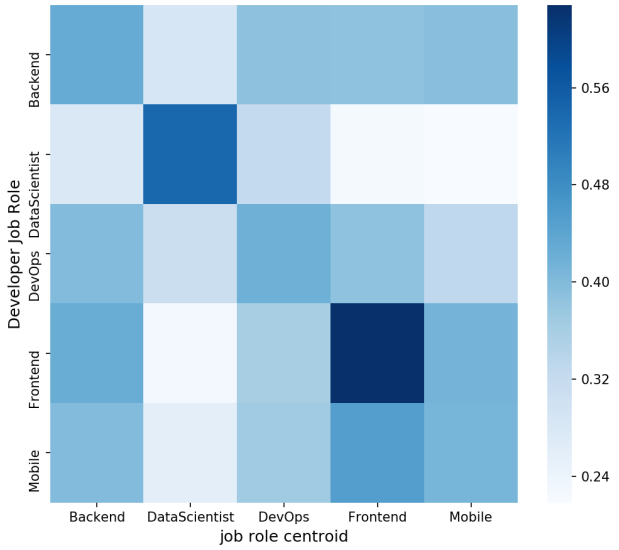


Figure 6.5 **Cosine Inter_Intra Topics (job roles).** Rows are the roles and columns are the role's centroid.

6.6 Threat to Validity

In this section, we explain the scopes and assumptions within our study that may pose a threat to the validity of our results.

Data Assumption

For *dev2vec:APIs*, we use a pre-trained `doc2vec` model on the API list collected from source files that were modified by developers with different commits. We use the same approach as [164] to collect the API calls from developers' commits. We assume that if developers submit a commit on a file, they have a basic knowledge about the libraries used in that source file [164]. With this assumption, we are not collecting the actual practice of developers. Because the developer may not change the list of APIs in a source file by submitting a commit. Therefore, the performance of embedding vectors derived from *dev2vec:APIs* to represent the expertise of developers can be affected by this assumption.

Further along, in *dev2vec:repos*, sometimes, developers with different domains of expertise contribute to the same repositories. Thus, we may collect the same content (the textual information of repositories) for developers with different domains who contributed to the same project. This assumption can impact the performance of embedding vectors derived

from *dev2vec:repos*.

Averaging embedding vectors of APIs

As we already discussed, in *dev2vec:APIs* due to the limitation of the pre-trained model [164], we calculate the average of the embedding vectors associated with APIs in a developer's list of API-related activities to represent the developer's expertise.

one naive solution could involve deriving a vector for each commit made by a new developer and then combining all these embedding vectors from various commits to construct a singular vector representing the developer's expertise. However, it's worth noting that the inference stage in `doc2vec` can yield different vectors for the same document in different attempts [33]. While the cosine similarity between these vectors tends to be high, they are not identical. To illustrate this, consider the scenario where two developers each submit different commits to the same source file without altering the list of APIs in that file. During the inference stage, the `doc2vec` model predicts two different embedding vectors for these two commits, even though they contain the same content. This slight discrepancy between embedding vectors derived from identical content can introduce noise into the final vectors of expertise.

There are advantages to our proposed method, *dev2vec:APIs*. First, we preserve the repetition of APIs within developers' activities by merging the API calls from source files across all commits. Second, we ensure that the same API is represented by the same vector in different commits and for different developers, as we retrieve the learned embedding vectors for each API from the pre-trained model. This averaging process can be considered as a weighted average. In other words, if a developer frequently uses two specific APIs in comparison to the others, the final embedding vector will exhibit greater similarity to the average embedding vectors of these two APIs. However, this averaging process may downplay the influence of less frequently used APIs (rare ones) in defining the expertise of developers. In future studies, alternative mechanisms, such as an attention mechanism, could be explored to learn weights for aggregating the API vectors, potentially resulting in a more accurate vector for representing developer expertise.

Type and source of activities

In this study, we concentrate on three prominent categories of developers' activities on GitHub. Our analysis reveals that the effectiveness of representing developers' domain expertise in embedding vectors is notably influenced by the source of these activities. It's essential to note that beyond the activities considered in this study, there exist various other activi-

ties on GitHub that can provide insights into developers' expertise. These activities include committing messages submitted by developers, code reviews, and the structural aspects of code authored by them. Moreover, GitHub is not the only platform containing records of developers' activities. Alternative platforms, such as StackOverflow, also harbor valuable information regarding developers' domain expertise. Future studies have the opportunity to explore these additional activities on GitHub and other platforms like StackOverflow, intending to acquire a deeper understanding and generate embedding vectors of expertise.

Job roles of developers

The pre-labeled dataset employed in this study categorizes a set of developers into five distinct job roles. However, it's important to acknowledge that developers' job roles extend beyond these five categories, but the research conducted by Montandon et al. [30] revealed that 64% of job postings on StackOverflow are associated with one of these five roles.

6.7 Chapter Summary

In this chapter, we delve into the representation of developers' domain expertise across their contributions to various software projects. While previous methods have shown promise in assessing developers' expertise within a single software project, they often prove impractical when applied to multiple projects due to the volume of data involved. To tackle this challenge, we introduce the *dev2vec* approach, built upon the *doc2vec* technique, which enables us to represent developers' expertise in an embedding space. We trained three distinct models using information sourced from GitHub, each capturing different aspects of developers' expertise. These models are referred to as *dev2vec:Repos*, *dev2vec:Issues* and *dev2vec:APIs*. Furthermore, we merge the output of these three methods by concatenating the embedding vectors from these three spaces, named as *dev2vec:RIAs*. To assess the effectiveness of our proposed methods in representing developers' domain expertise, we employ these embedding vectors in a significant problem within the software engineering domain: classifying developers according to their job roles. Our results in this classification task demonstrate notable improvements compared to state-of-the-art approaches.

CHAPTER 7 ASSESSING EXPERTISE OF DEVELOPER FROM THE STATISTICAL DISTRIBUTION OF PROGRAMMING SYNTAX PATTERNS

7.1 Chapter Overview

As the quality of code generated by automated tools impacts developers' productivity and also their contributions to software projects, the accurate assessment of developers' programming expertise is also crucial for assigning the right candidates to perform a task or, more broadly, involving them in a project that requires a sufficient level of knowledge. Potential programmers can be drawn from a large pool. Therefore, automatic means to assess their expertise from written programs would be highly valuable in such a context. Previous research aimed at this goal has often relied on heuristics like the "Line 10 Rule" or linguistic information in source files, such as comments or identifiers, to represent developers' knowledge and evaluate their expertise. In this chapter, we focus on mastering syntactic patterns as evidence of programming expertise. We propose a theoretical definition of programming knowledge based on the distribution of Syntax Patterns (SPs) in source code, namely, Zipf's law. We evaluate the performance of this model in classifying developers into two categories of "Expert" and "Novice".

7.2 Study Design

In this section, we establish the objectives of this chapter. Chapter 2 provides a brief background on the Zipf distribution which is a pivotal concept to the proposed approach. Figure 7.1 provides an overview of our approach for representing developers' expertise using syntactical patterns through Zipf's law and classifying developers into categories of "Expert" and "Novice". Each step will be elaborated upon in the Approach section, Section 7.3.

7.2.1 Setting Objectives

Identifying the expertise of developers is important when searching for an individual to hire, to solve a bug, or to contribute to a software project. The cost of poor recruiting decisions is estimated to be as high as ten times the annual salary of an employee [11]. While platforms such as GitHub offer a valuable information source about an individual's skills [230,231], non-technical recruiters have difficulty assessing applicants based on their GitHub activities [232].

Representing a developer's knowledge from their written code is challenging. Previous works

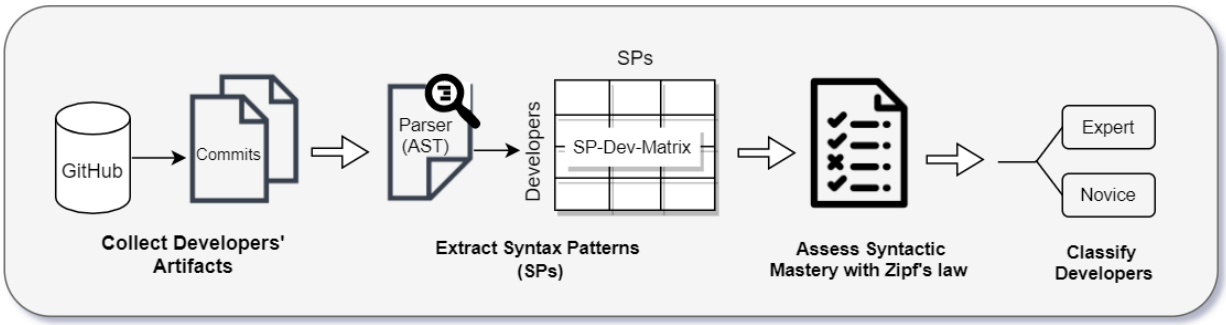


Figure 7.1 The process of the proposed approach in classifying developers based on syntactic mastery

have relied on heuristics such as the *Line 10 Rule* and the number of commits in different source files [25–27], or the number of different API calls [28,29] to represent the knowledge of developers. Another group of studies analyzes textual and linguistic information within code, like comments or function names, to represent the domain knowledge of a developer [14,136]. However, many of these aforementioned heuristics provide indirect evidence of knowledge and can prove unreliable or biased indicators of expertise.

In this chapter, our focus is on mastering programming language syntax patterns as a proxy of programming expertise. We distinguish expert developers by evaluating their proficiency in using programming syntactic patterns.

We assume that a developer's knowledge comprises a subset of programming constructs they have mastered, including syntactic patterns and lexical expressions present in their artifacts. This idea is also inspired by a linguistic theory known as Zipf's law [34], which represents the distribution of words in a corpus. Baixeries et al [159] observed that an individual's communication skills are reflected in the Zipf distribution of their vocabulary in conversations.

The model we propose relies on the Zipf distribution of syntactic patterns found in artifacts created by developers to assess their mastery of programming syntax patterns and to distinguish experts from novices. Evaluation of our proposed model using real data demonstrates that it outperforms various baselines and state-of-the-art techniques in classifying experts from novices. Therefore, this model represents an important initial step toward our community's long-term goal of developing an efficient technique for automatically assessing developers' expertise.

This chapter offers the following contributions:

- We introduce a model for evaluating developers' proficiency in programming syntactic patterns through parameter estimation of Zipf's law.

- We classify expert developers from novices based on their mastery of using syntax patterns, achieving better performance compared to recent state-of-the-art methods.

To address the objectives of this study, we formulate the following research questions:

- **RQ1:** How does expertise impact the distribution of SPs in developers' commits?
- **RQ2:** To what extent can Zipf's law effectively evaluate developers' expertise in programming SPs and distinguish experts from novices?

7.2.2 Background Overview

Zipf's law is known to model the word distribution in natural languages, including English [67]. More recent research has revealed that Zipf's law also applies to programming languages such as the distribution of lexical tokens in Java, C++, and C source code [65]. Furthermore, several studies have compared the Zipf distribution in human language and programming languages [153, 154], revealing differences in the range of values for the α parameter in Zipf's law between the two.

Additionally, studies have explored the relationship between expertise and Zipf's law. One such study estimated vocabulary growth in second-language learners using Zipf's law [233]. For example, beginners in speaking English may use simple phrases like "I am angry because I cannot fix this problem" before progressing to more advanced sentences like "I find this problem frustrating". Another study compared the speech of children and adults in terms of the Zipf distribution, demonstrating a connection between Zipf's law parameters and individuals' communication skills [159].

In this chapter, we delve into a similar relationship between the parameters of Zipf's law and the distribution of syntactic patterns in developers' code artifacts, leveraging it to assess their mastery of programming syntax patterns.

7.3 Approach

In this section, we provide a detailed description of each step in our proposed approach, as illustrated in Figure 7.1.

7.3.1 Collect Developers' Artifacts

A program is a product of many contributors. Version control systems like GitHub enable us to access programmers' artifacts from various repositories over time. The aggregation of a

developer's commits across different projects on GitHub serves as a representative sample of their experience. We gather these commits on source files (i.e., all files with a ".py" extension) across their contributions to different projects to establish their knowledge state.

7.3.2 Extract Syntax Patterns

In the theory of knowledge space, an individual's knowledge state is defined as a subset within a knowledge domain [234]. In this study, this domain is defined as a set of syntactical patterns.

```
1 def plus(x):
2         return x+2
3
4 output= []
5 for i in range(1,5):
6         output.insert(i,plus(i))
7
8 print(output)
```

(a)

```
1 print(list(map(lambda x: x+2, range(1,5))))
```

(b)

Figure 7.2 (a) and (b) are two different methods of adding a variable (between 1 to 5) with 2, in Python

Each commit comprises various elements, such as API calls, identifiers, lexical expressions, and programming language keywords. The manner in which developers employ programming language keywords and expressions results in the creation of different Syntax Patterns (SPs) within their artifacts. Developers may employ different approaches and, consequently, different SPs to address a problem based on their knowledge state. For instance, in Figure 7.2, we can observe two distinct solutions to solve the same problem in the Python language. In this research, we gather programming keywords and lexical expressions from commits as the knowledge items attributed to their respective authors. We utilize the Abstract Syntax Tree (AST) to extract and collect these SPs.

Abstract Syntax Tree (AST): An Abstract Syntax Tree (AST) serves as an abstract representation of a parsing tree [235]. The AST exhibits a larger vocabulary size compared to the structure of the code. It introduces new nodes within the AST that symbolize the

relationships between abstract components in the programming code [236]. For instance, Figure 7.3b shows the AST representation of the Python code in Figure 7.3a. We extract the class names and some of the attributes of different nodes from the AST as SPs. Figure 7.3c displays the SPs we extract from the sample code in Figure 7.3a. In this figure, for example, {*'Subscript', 'Index[Name]'*} denotes accessing a subscript with a variable indexing type. We exclude the lowest layer in each path of the AST, which typically consists of variables or function names (textual information). While the lowest layer may convey project-specific knowledge unique to different repositories and domains, we have chosen to confine the scope to more generalized knowledge items because our objective is to assess expertise in using SPs.

```
def dt(i):
    x = Data[i]
    return x
```

(a) Sample python code

```
FunctionDef(
        name='dt',
        args=arguments(
                    args=[arg(arg='i', annotation=None)]),
        Assign(
            targets=[Name(id='x', ctx=Store())],
            value=Subscript(
                        value=Name(id='Data', ctx=Load()),
                        slice=Index(value=Name(id='i', ctx=Load())),
                        ctx=Load()))],
      returns=None)])
```

(b) AST dump

```
['FunctionDef', 'arguments: Standard', 'Assign', 'Name', 'Subscript', 'Index[Name]',
'Return']
```

(c) Syntax Patterns

Figure 7.3 AST and a list of Syntax Patterns for a sample Python code. (a) shows a sample code in python, (b) is the AST of sample code in (a), and (c) represents the Syntax Patterns (SPs) collected from AST in (b)

For each commit in Git, there is a file named *diff*, which contains all the lines that a developer modified in a source file. However, this *diff* file doesn't specify the exact changes made by the developer. To overcome this limitation, we employ a technique where we compare the

Abstract Syntax Tree (AST) of a source file before and after applying a commit [28]. We then gather the nodes that represent the differences (added or removed) between the two trees as knowledge items attributed to the author of the commit. This method ensures that developers have indeed applied the SPs we are collecting as part of their knowledge. Upon completing this step, we construct a matrix that correlates developers with the frequency of various SPs in their commits.

7.3.3 Assessing Syntactic Mastery with Zipf's law

In this section, we explore the relationship between syntactic mastery and the exponents of Zipf's law. Figure 7.4 illustrates the probability distribution of SPs in the commits of two developers with different knowledge states in Python. While both charts depict a Zipf distribution characterized by a small number of high-probability SPs and a long tail of low-probability SPs, they do not mirror the same behaviors. This indicates that the number of SPs and their distribution vary between these two developers.

For instance, the probability of the top-ranked SP denoted as $r = 1$, is approximately 0.07 for the expert sample and 0.05 for the novice sample. It's important to note that these may not necessarily be the same SPs. In addition, The expert sample reveals a longer tail, indicative of more advanced patterns, while the novice sample features a higher frequency of mid-ranking SPs.

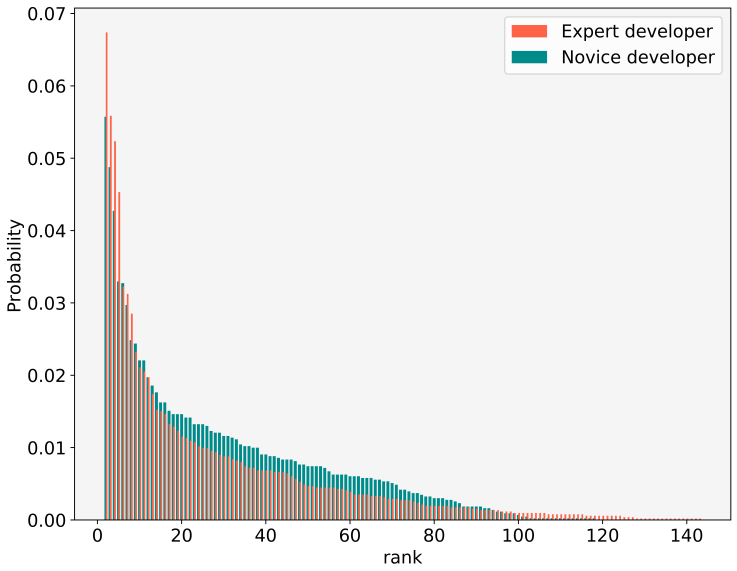


Figure 7.4 Probability distribution of SPs of two developers with different knowledge states

What is the relation between Zipf's law and Mastery of SPs?

Let's illustrate the connection between Zipf's law and programming syntax mastery with an example. Suppose we have two developers, denoted as dev_1 and dev_2, both possessing the same level of expertise in programming syntax and using similar SPs with the same frequency in their commits. When we apply Zipf's law to model the distribution of SPs in their commits, we obtain $\alpha = 1.072$ for both developers.

Now, let's consider a scenario in which we ask both developers to write code to solve a new problem, such as creating a function to add numbers within the range of 1 to 5 with a step of 2. Developer dev_1 chooses the approach in Fig. 7.2a, introducing three new SPs, namely {*'Add','List','insert'*}, while reusing most of their existing knowledge items. In contrast, dev_2 adopts the method in Fig. 7.2b and introduces more new knowledge items, including {*'map','Lambda','Add','List'*}.

After fitting the new distribution of SPs for each developer with Zipf's law, we find that the parameter α decreases to $\alpha = 1.02$ for dev_1 and increases to $\alpha = 1.19$ for dev_2. In Fig. 7.5, the bars represent the actual distribution of SPs, and the lines depict the fitted Zipf's law for these distributions. It is evident that different values of the exponent α are assigned to fit different distributions with Zipf's law. Since developers choose different SPs based on their knowledge when writing code, we can infer that there is a relationship between developers' mastery of SPs and the exponent α.

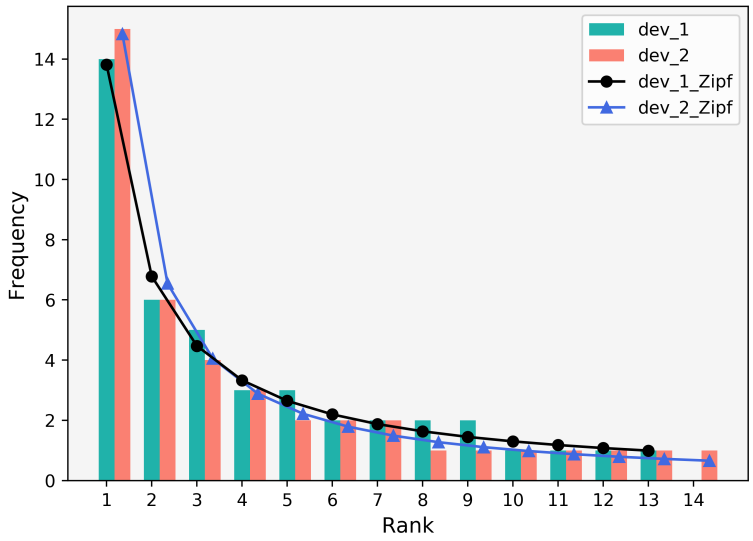


Figure 7.5 The Zipf distribution between SPs of two sample developers

7.3.4 Classify Expert

We focus on two distinct categories of developers: "Experts" and "Novices". "Expert" developers exhibit a high degree of expertise in programming, effectively utilizing SPs and resolving programming issues. In contrast, "Novice" developers have made fewer contributions and predominantly rely on basic SPs. We distinguish between experts and novices based on their level of SP mastery.

To achieve this, we fit the distribution of SPs in developers' commits using Zipf's law and estimate the α value for each developer. Subsequently, we employ this exponent to classify individuals into the expert and novice categories. This approach is referred to as *"zipf_α"*.

We set up two types of classification tasks: one where the proportion of experts and novices is unknown, and another where this proportion is determined from a labeled training set. These tasks correspond to two assessment methods: *"Clustering"* and *"Classifiers"*, respectively.

Clustering In the clustering group, we employ Jenks Natural Breaks Clustering (JNBC) [237, 238]. This method is suitable for single-dimensional data, such as our proposed method, *"zipf_α"*, which relies on a single feature_the estimated value of α_to represent syntactic mastery.

Classifier In the classifier group, given that this is a binary classification problem, we determine the optimal threshold for the exponent α to classify experts from novices directly from the training set [239].

7.4 Evaluation

In this section, we start by explaining the experimental setup of our study. Then, we present our results that address the following two research questions:

RQ1: How does expertise impact the distribution of SPs in developers' commits?

RQ2: To what extent can Zipf's law effectively evaluate developers' expertise in programming SPs and distinguish experts from novices?

7.4.1 Experimental Setup

In this section, we provide an overview of our dataset, followed by an explanation of how we fit a distribution using Zipf's law and estimate the goodness of fit. Afterward, we present

a list of methods against which we assess the performance of our proposed model in expert identification.

Dataset

To demonstrate the characteristics of our model, we utilize both synthetic and real datasets. Considering that the real data lacks labels (ground truth class) and manually inspecting the expertise of a large number of developers is expensive and time-consuming, we generate a labeled synthetic dataset comprising 1200 developers. This dataset is derived from real sample commits of developers. We use this synthetic data to calibrate the differences in SP distribution within the code of a substantial number of developers. Additionally, we used it to determine the optimal number of developers that should be subject to manual inspection within the real dataset.

Collect Real Data We collect developers' commits on source files in GitHub to assess their knowledge state. Our primary focus is on the Python programming language, as it is widely used across various fields today. Python has gained popularity among individuals from diverse backgrounds, including those without prior programming or software development experience, due to the importance of Artificial Intelligence (AI) in various domains.

For collecting GitHub data, we employ the PyDriller tool [240]. We apply the following criteria to collect data from developers commits with varying levels of expertise. A summary of the collected data is presented in Table 7.1.

1. Begin by selecting the top 10 Python projects on GitHub, ranking them by the number of stars they've received.
2. Identify the top 10 contributors with the most commits and the bottom 10 contributors in the list for each repository.
3. Create links for each author's name, directing to their GitHub profile, based on their aliases or usernames.
4. For each repository in their profile, collect commits made in the last year. In cases where a developer's data doesn't meet the optimal sample size according to Zipf's law, we collect more commits over multiple years until we reach the optimal size.
5. In addition, select the top 50 and the bottom 50 Python developers on GitHub based on the number of commits they've made in Python programming language projects. This ensures a more diverse representation of developers and projects.

6. Repeat step 4 for each of the 100 developers identified in step 5.

Table 7.1 A summary of the dataset collected from GitHub

# of Commits	# of Projects	# of Developers
54676	441	300

We collected a total of 54,676 commits. For each commit, we compared the AST of the source file before and after the commits were applied, resulting in the discovery of 197 unique SPs.

Generate Synthetic Data We generated a labeled dataset with two categories: *"Expert"* and *"Novice"*. Here's how we achieved this:

First, we manually selected an expert developer from our real developer dataset. We then calculated the probabilities of all SPs within this expert developer's commits. These probabilities served as weights to generate SPs in the commits of synthetic *"Expert"* developers. For SPs that were absent in the sample of the *"Expert"* developer, we applied an uninformative prior, using the probability of those patterns across the entire dataset.

We repeated this process with 5 and 10 expert samples from the real dataset. Subsequently, we conducted a chi-square test to assess whether the three synthetic datasets generated based on 1, 5, and 10 expert samples were significantly different. The results of the test indicated that we couldn't reject the null hypothesis, suggesting that these three synthetic datasets shared a common distribution and were not significantly different.

We then repeated this process with a sample of *"Novice"* developers to generate synthetic *"Novices"*. To explore the scalability of the model, we generated 600 *"Expert"* and 600 *"Novice"* developers.

Fitting Zipf's law

As suggested in [155], the maximum likelihood method is recommended for estimating α. Let's assume we have a set of n SPs: $\{sp_1, sp_2, \ldots, sp_n\}$, and that $Pr(sp; \alpha)$ represents the probability distribution function of SPs. If the probability of sp_i, is denoted as $Pr(sp_i; \alpha)$, then the maximum likelihood $L(\alpha)$ is defined as the joint probability of each sp_i:

$$L(\alpha) = \prod_{i=1}^{n} Pr(sp_i; \alpha) \tag{7.1}$$

We assume that all n data points are independent and maximize the log-likelihood of $L(\alpha)$ [157, 158]. We can replace $\log Pr(sp_i; \alpha)$ with (2.3) in the log-likelihood function and estimate the parameter α in Zipf's law with (7.2):

$$\log L(\alpha) = -\alpha \sum_{i=1}^{n} \log r_i - \sum_{i=1}^{n} \log H_{n,\alpha} \tag{7.2}$$

To assess the goodness of the fit, we employ the Kolmogorov-Smirnov (KS) test and refer to the KS table for power law distribution [155] to estimate the goodness of fit.

The optimal sample size for fitting a distribution with Zipf's law is the saturated Harmonic series $n * H_n$ [63]. In our study, n represents the number of distinct SPs. Thus, in our specific case with 197 unique SPs, the optimal sample size is 1156 data points per developer.

Comparable Methods

In this section, we discuss different baselines and methods that we used to assess the effectiveness of our approach. Initially, we explain how these methods capture developers' knowledge, followed by an explanation of their techniques for expert identification.

Represent the Knowledge As our first baseline, we construct a feature-based model inspired by prior research. Given that our approach to representing developers' knowledge in a programming language centers around the mastery of SPs, which differs from the case studies in previous works (like identifying experts in specific libraries [29]), a direct comparison with the models proposed in those prior studies is not feasible. Consequently, we opted to create a new baseline employing the features utilized in state-of-the-art approaches from the existing literature.

A group of features commonly used in previous studies pertains to the *Quantity* of changes made by developers, encompassing metrics like the number of commits or lines of code [25, 26, 241]. Another set of features focused on the *Frequency* of activities, including the time intervals between commits or the recent date of commit submissions [27, 230]. The final cluster of features captures the *Breadth* of contributions, as assessed by metrics like the number of projects [242]. In a previous study [15], authors calculated correlations between various features to identify experts. We replicated their model using the features summarized in Table 7.2. In the subsequent discussion, we refer to this model as baseline *bl_ft*.

As a second baseline, inspired by the study of Teyton et al [138], we employ a vector space technique to represent a developer's knowledge based on the frequency of different SPs with-

Table 7.2 Features collected to represent knowledge in the aseline *"bl_ft"*

Feature Name	**Description**
numCommits	Number of commits in Python projects
LOC	Number of added and removed lines
avgInterval	Average interval between commits
lastCommit	Number of days since last commits
numProject	Number of Python projects that a developer has contributed

out fitting the distribution between them to Zipf's law. We calculate the*tf-idf* of SP frequency in developers' commits and use it to depict their knowledge. This approach allows us to examine whether representing developers' knowledge with syntax patterns has any impact on the identification of experts in comparison to modeling this knowledge with quantitative features, such as the number of commits. We refer to this baseline as *bl_vs*.

As the final baseline, we construct a naive baseline that represents each developer's knowledge by the count of unique SPs in their commits. We name this baseline *bl_sp*.

Since the proposed model, *Zipf_*α, relies on single-dimensional data, we merge it with the *bl_ft* baseline and include the exponent α as an additional feature in the list of features used in the *bl_ft* baseline. This hybrid approach assesses the impact of combining quantitative features with the distribution of SPs on the representation of developers' programming knowledge. To implement this combination.

Classify Experts To identify experts, we employ both Clustering and Classification techniques, as previously explained in Section 7.3.4 for the *Zipf_*α approach. In the clustering setup, we utilize Jenks Natural Breaks Clustering (JNBC) [237, 238] for single-dimensional data, *bl_sp*. Alternatively, for methods with multi-dimensional data points, like *bl_ft*, we apply the k-means clustering algorithm. In our case study, we set $k = 2$ since we are focusing on two categories: *Experts* and *Novices*. In the classification setup, we choose between random forest or SVM, depending on the number of dimensions for multi-dimensional data. For single-dimensional data, we determine the optimal threshold.

Table 7.3 displays a list of all methods assessed in this paper, including both our novel methods and baselines.

Table 7.3 A description of the different methods used to identify experts.

Method	Description
bl_ft	A baseline that employs the features listed in Table 7.2 to represent developers' knowledge.
bl_vs	A novel baseline that uses the vector space of developers and the frequency of SPs in their commits (SP-Dev-Matrix).
bl_sp	A baseline that uses the number of unique SPs as an indicator of expertise.
zipf_α	The proposed model, which uses the exponent α as a factor for expert identification.
zipf_α_ft	Combination of *zipf_α* and *bl_ft*.

Assessing the Performance of the Models

Cross Validation We employ Leave-One-Out Cross-Validation (LOOCV) to assess the predicted categories in both setups. In the context of clustering techniques, LOOCV involves isolating one developer as a test case at each iteration and applying the clustering algorithm to the remaining developers as a training set. We designate the cluster with the highest number of experts as the *Expert* cluster. Subsequently, we calculate the distance between the feature(s) of the test case and the centroid of each cluster and predict the appropriate category by selecting the cluster with the minimum distance.

Evaluation Metrics We assess the results by calculating precision, recall, and F1-score for each category, *Expert* and *Novice*, individually. Additionally, we report the Kappa statistic, considering both categories.

7.4.2 Experimental Results

In this section, we discuss our findings on addressing the research questions.

RQ1: How does expertise impact the distribution of SPs in developers' commits?
As we discussed earlier in Section 7.3, two developers with two different knowledge states choose different sets of SPs to solve the same programming problem. Writing different artifacts such as commits generates a distribution between SPs in the knowledge state of a developer. This distribution follows Zipf's law. We noticed that to fit different distributions with Zipf's law, different values are assigned to the exponent α in Zipf's law.

To address research question *RQ1*, we fitted the data generated for developers in two categories: *"Expert"* and *"Novice"*, into the Zipf distribution. We applied Zipf's law to fit the distribution of SPs in the commits of each developer, separately, obtaining the value of the

exponent α for each developer. Figure 7.6 illustrates the distribution of α in the two categories of *"Expert"* and *"Novice"*. We discovered that there is no overlap in the range of α values between developers in these two categories. This indicates that developers' proficiency in SPs influences the range of α required to fit the distribution of SPs in their commits with Zipf's law. The mean of α is 0.75 for *"Novice"* developers and 0.96 for *"Expert"* developers.

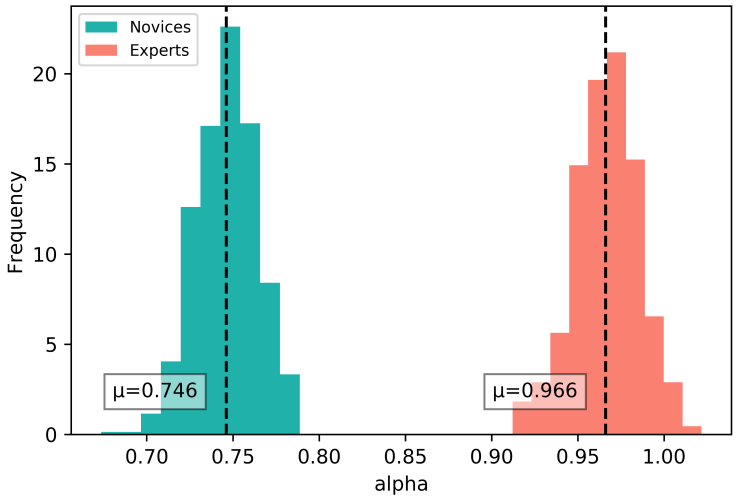


Figure 7.6 Distribution of exponent α for developers in synthetic Data

To determine the optimum number of developers to be manually labeled from the real dataset, we conduct a hypothesis test to find the optimal sample size. Referring to Figure 7.6, which displays distinct means for α in the two categories of *"Expert"* and *"Novice"*, we formulate a hypothesis and apply a t-student test with a 95% confidence level to identify the optimal data size:

H_0: *"The means of the distributions for the exponent α are equal between novices and experts".*

H_a: *"The means are different between novices and experts".*

According to the test results presented in Table 7.4, the optimal number of developers to reject the null hypothesis (H_0) is 20. Consequently, we manually labeled 20 *"Expert"*s and 20 *"Novice"*s.

Manual Labeling: Two authors, acting as "expertise evaluators", independently performed an assessment of 20 *"Expert"* and 20 *"Novice"* developers from the real dataset [15,136,138]. One of the evaluators is a Ph.D. candidate in software engineering, and the second is an MSc. student, also in software engineering, both with several years of professional working experience. The evaluators reviewed various aspects, including commits, pull requests, and code reviews submitted by the developers in GitHub Python projects, continuing until they felt confident in their assessments of the developers. They examined a random sample of de-

velopers' contributions, ranging from 10% to 30%, for those with more than 500 contributions in their GitHub profile. For individuals with fewer than 500 contributions, the evaluators examined 40%, 70%, or all of the developer's commits. They also checked individuals' profiles on Stackoverflow.com if they had one and reviewed random Q&A samples in which they were involved. The same procedure as on the GitHub platform was applied to individuals with both more and fewer than 500 reputation points on Stackoverflow. Additionally, they checked individuals' profiles on Linkedin.com when possible, verifying their career information and lists of skills. To ensure that both evaluators shared the same perspective during the manual inspection, they employed two different classification procedures: the first evaluator classified developers into two categories, *"Expert"* and *"Novice"*, while the second evaluator assigned ratings ranging from *"Novice"* to *"Expert"* with labels of 1, 2-, 2+, and 3.

Table 7.4 Hypothesis testing to determine the optimal number sample size of developers

t-statistic	**p-value**	**sample size**
-45.76	<<0.0001	20 (20 novices and 20 experts)

The lowest level on the scale indicates that the individual has contributed to Python repositories, including individual repositories. However, their contributions do not extend to functional aspects such as code reviewing or adding functionality, optimization, testing, or bug resolving in high-star projects within the Python repositories. They may not have answered questions related to Python programming on Q&A platforms. The highest level signifies that the individual has made various functional contributions to significant Python projects, answered questions related to Python programming and/or its libraries on Q&A platforms, and may have a relevant career background in Python programming according to their LinkedIn profile.

The two middle scales, 2- and 2+, pertain to individuals who fall into the lowest or highest levels but cannot be categorized as such. The labeling team compared the provided labels with these defined scales and calculated the agreement percentage between the two evaluators using Cohen's Kappa [1] and achieved an 87.5% agreement. The disagreements were observed in five cases. For instance, Dev2 contributed to eight Python projects with a total of 1384 commits, and he is an AI researcher and postdoc student. The first evaluator categorized him as "Novice", while the second evaluator assigned him a scale of 2+. In another example, Dev5 had 316 commits in 36 projects and was identified as a Python developer based on his LinkedIn page, but his contribution metric on GitHub was 43. The first evaluator labeled

[1] http:// vassarstats.net/kappa.html

him as *Expert* based on the content of his commits, while the second evaluator scaled him as 2+. We kept the cases of disagreement in the dataset with both labels and conducted all evaluation metrics in RQ2 twice, each time based on the manual inspection of one of the evaluators, and then reported their average.

Following the labeling step, we proceeded to fit the distribution of SPs in the commits of these developers with Zipf's law. As illustrated in Figure 7.7, this figure showcases the distribution of α for the selected sample of 40 real developers. In alignment with our observations in synthetic data, we noticed that the mean of α values differ between these two categories, even though there is some overlap in the ranges of α.

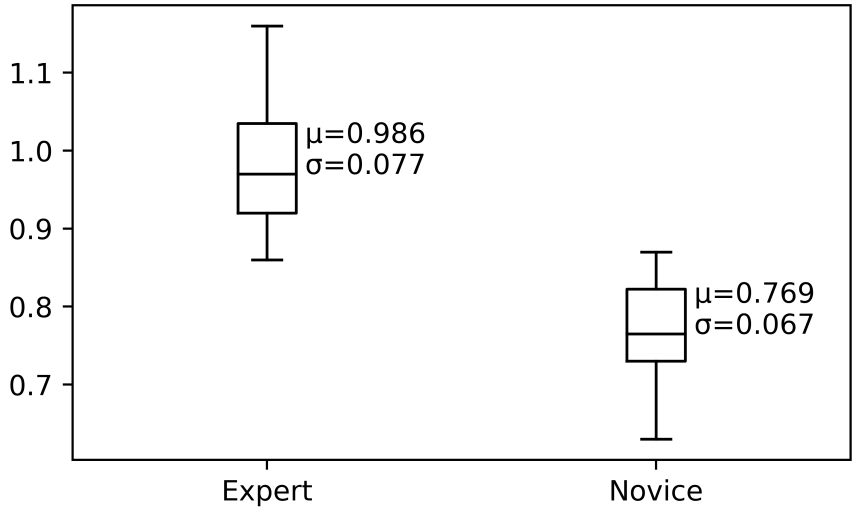


Figure 7.7 Distribution of exponent α between developers in real data

Finding 1: The syntactic mastery of developers influences the distribution of Syntax Patterns (SPs) in their commits, and the exponent α in Zipf's law serves as an indicator of this mastery.

RQ2: To what extent can Zipf's law effectively evaluate developers' proficiency in programming SPs and distinguish experts from novices?

In our second research question, we aim to assess the performance of our proposed approach compared to the baselines in distinguishing *Expert* developers from *Novice* ones.

Table 7.5 displays the results for both clustering and classifier setups. The methods with single-dimensional features, *zipf_*α and *bl_ps*, yield the same results in both groups of clus-

tering and classification. Our proposed method, *zipf_α*, outperforms all the baselines. It achieves a precision, recall, and F1-score of 89.74%, 85.48%, and 87.5%, respectively in both setups when categorizing *Exper*. The method *zipf_α_ft*, which combines quantitative features and the content of changes (a mixture of *zipf_α* and *bl_ft*), exhibits the highest performance in the classifier setup, with precision, recall, and F1-score of 92.5%, 90.39%, and 91.4%, respectively.

The feature-based baseline, *bl_ft*, and the vector space baseline, *bl_vs*, demonstrate better performance compared to the naive baseline, *bl_sp*, in both setups. This indicates that the number of distinct SPs alone is not a reliable indicator of developers' expertise. Methods with multi-dimensional features, such as *bl_ft*, *bl_vs*, and our proposed method *zipf_α_ft*, exhibit better performance in the Classifier setup.

For instance, Dev8, categorized as an *Expert* (or scale 3) by both evaluators, was wrongly predicted as a *Novice* in the clustering setup but was correctly identified as an *Expert* using the classification technique. Dev8 has 1083 commits, and contributed to 16 Python projects, with his last commit made 26 days before our data collection date. He added 10,499 lines of code to different projects over the past year and is one of the main developers of a GitHub repository with 36,000 stars. After applying k-means, the distance between this developer's feature vector and the centroids of the *Expert* and *Novice* clusters is 0.35 and 0.33, respectively, in the normalized dataset of the *bl_ft* method. Thus, Dev8 is categorized within the cluster with a minimum distance of 0.33. Another example, Dev15, was incorrectly classified as a *Novice* with *bl_ft* in the Classifier setup, even though both evaluators agreed to classify him as an *Expert*. However, when using the *zipf_α_ft* method in the Classifier setup, he was correctly identified as an *Expert*. This result demonstrates that combining syntactical patterns with quantitative features, such as the number of commits, enhances the performance of a binary classifier in identifying programming experts.

Finding 2: Our proposed method, *"zipf_α"*, achieves a higher precision than other baseline methods for identifying *"Expert"*, in both Clustering and Classifier setups. Combining quantitative features with the distribution of SPs, as in the *zipf_α_ft* method, results in a significant performance improvement of up to 18.45%.

7.5 Discussion

This chapter delves into the demonstration of a theoretically grounded technique for representing the expertise of developers in a programming language. While this study marks

Table 7.5 Results on real data in two setups of clustering and classifier

	Clustering					**Classifier**				
	Proposed methods		**Comparable Methods**			**Proposed methods**		**Comparable Methods**		
metric/method	**zipf_α** (JNBC)	**zipf_α_ft** (k-means)	**bl_ft** (k-means)	**bl_sp** (vocab size) (JNBC)	**bl_vs** (k-means)	**zipf_α** (threshold)	**zipf_α_ft** (random forest)	**bl_ft** (random forest)	**bl_sp** (vocab size) (threshold)	**bl_vs** (SVM)
Precision%(Exp)	**89.74**	89.68	67.93	47.37	68.12	89.74	**92.5**	70.5	47.37	77.5
Precision%(Nov)	85.48	**86**	66.25	45.24	69.81	85.48	**90**	72.14	45.24	75
Recall%(Exp)	85.48	**85.6**	70.83	43.93	73.33	85.48	**90.36**	75.6	43.93	75.71
Recall%(Nov)	89.74	**89.74**	63.16	48.68	64.08	89.74	**92.24**	66.58	48.68	76.84
F1-score%(Exp)	**87.5**	87.37	69.32	45.58	70.54	87.5	**91.4**	72.95	45.58	76.59
F1-score%(Nov)	87.5	**87.62**	64.62	46.89	66.71	87.5	**91.09**	69.23	46.89	75.9

an initial step toward the goal of developing an efficient method for automatically assessing developers' expertise, its findings have consistently outperformed different baselines and state-of-the-art techniques.

The findings in this chapter reveal that developers' expertise impacts the distribution of programming syntax patterns in their code. This distribution adheres to Zipf's law, and the exponent, α, can be influenced by variations in this distribution. Our findings further show that using this exponent to classify developers into two categories, experts and novices, performs better than both baselines and state-of-the-art methods. Additionally, combining quantitative metrics such as the number of commits or lines of code with our proposed method, which reflects programming syntax mastery, can improve the accuracy of identifying experts by up to 18.45%.

Regarding zipf_α, the proposed method holds an advantage as it does not require a training process and it acts as a threshold to identify experts. However, when integrating zipf_α with a feature-based approach (bl_ft) derived from state-of-the-art methods, we employ a classifier (random forest) and also a clustering technique (k-means) with Leave-One-Out Cross-Validation (LOOCV) to identify experts (zipf_α_ft).

It's important to note that our focus in this chapter is on the Python programming language and the extraction of syntax patterns from the AST of the Python code. Since the parsers and programming structures of different programming languages vary, we cannot apply the same syntax patterns to other programming languages. However, our method for extracting syntax patterns can be adapted to other programming languages by considering their respective AST structures and following the remaining procedure to fit the distribution of SPs into Zipf's law.

Previous studies, inspired by the "line 10 rule", considered developers as experts in a file if they made changes to it. However, their method was limited to representing developers' expertise within a single software project and couldn't be extended across different software

projects. We are one of the first studies to propose a method that is not confined to a single project and can be applied both within and across various software projects.

While this chapter specifically focuses on programming syntax mastery as evidence of developers' expertise in a programming language, further investigation, such as technical assessments or personal interviews, remains necessary to select the final expert. Our method serves as a filter for extracting expert developers from the vast pool of developers expert in a programming language, and we acknowledge that developers' expertise encompasses more aspects beyond syntax mastery.

7.6 Threats to Validity

The accuracy of estimating exponent α is influenced by two factors: "Number of SPs" and "Size of Corpus". Besides, the performance of our model depends on the "Validity of Data in GitHub". In this section, we discuss how these factors pose threats to the validity of our experiments.

Number of SPs and Size of Corpus

In this study, we focused on the Python programming language. In the real dataset, the number of SPs obtained after applying AST to developers' commits is 197. However, this number may vary in other programming languages. It's important to note that not all developers use all of these 197 SPs in their code, and the usage of specific SPs may be related to the domain of programming tasks they are addressing.

The accuracy of estimating the parameter α in Zipf's law depends on both the minimum number of SPs and the corpus size. To illustrate the dependency on the minimum number of SPs, we generated four synthetic datasets with Zipf distribution while $\alpha = 1, 0.9, 0.8$, and 0.7. The corpus size ($T^* = 1156$) and the number of SPs (equal to 197) were kept constant. We then calculated a confidence interval for each value of the exponent α.

To determine these confidence intervals, we employed a bootstrap method with replacement. For each distribution, for example, $\alpha = 0.9$, we randomly selected a sequence of SPs and generated 100 different sample datasets from the main population. We estimated α for all 100 sample datasets and calculated the confidence interval for α. To assess the model's dependency in estimating α as a function of the number of unique SPs, we varied the number of SPs from 197 to 5 and estimated the parameter α in the new distribution.

Figure 7.8 displays the estimated α for different numbers of SPs. For example, for $\alpha = 0.9$, if the number of SPs (or rank in Zipf's law) is less than 66, the predicted α falls outside the

confidence interval. We can observe that when the number of SPs is less than approximately 63 on average, the estimated α is outside the confidence interval.

Another factor that can impact the estimation of the exponent α is the corpus size, which refers to the total number of SPs with frequency in the commits of developers. As discussed in Section 7.4.1, the quantity $n \cdot H_n$ indicates the optimum corpus size. In our specific case study, involving 197 distinct SPs, the optimal corpus size is determined to be $T^* = 1156$. The Residual Standard Error (RSE) is less than 1% for a corpus size greater than 1156.

In our case study, the proposed model is unable to make a reliable decision if a developer uses less than 63 distinct SPs in their commits or if the corpus size is less than $T^* = 1156$. This can be viewed as a *cold start* issue for our method. To mitigate this problem, we exclude developers with less than 63 distinct SPs or a corpus size of less than 1156.

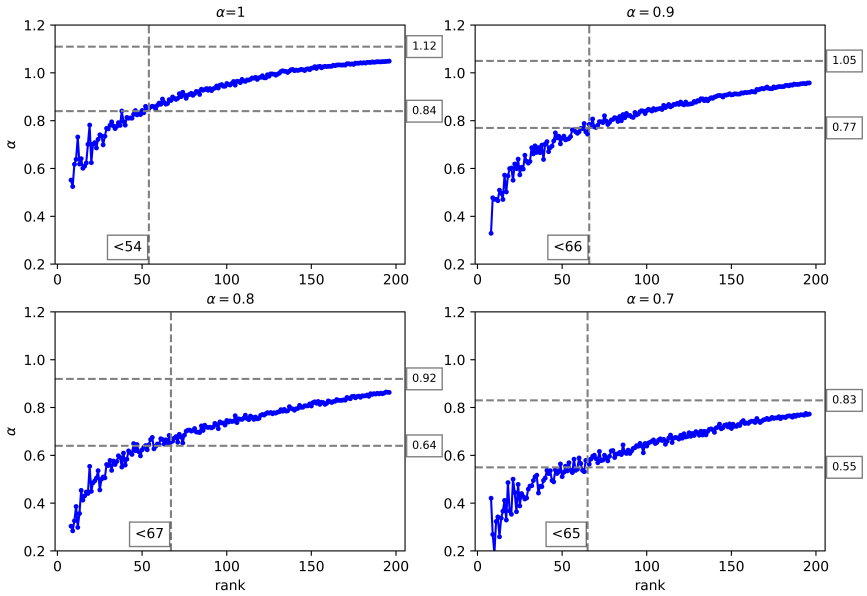


Figure 7.8 Minimum number of SPs to fit Zipf's law and the confidence interval of α in $\alpha = 1, 0.9, 0.8$ and 0.7

Validity of Data on GitHub

In our dataset, we collect the contributions of developers in public GitHub projects. However, there are developers on GitHub who have sparse public contributions but make more contributions to private projects. We lack access to the data from these private projects, which could have provided additional information for those developers, thereby mitigating the cold start issue. The availability of such information from private repositories could also influence the categorization of developers in the *Novice* category.

Another potential threat to validity arises from our assumption that all commits are authored by the individuals whose names are specified as commit authors. We did not account for the potential impact of code copying or the use of code generation tools in the commits of developers.

Also, we primarily focus on developers' GitHub commits to extract evidence of their expertise, yet it's important to note that there are additional sources of knowledge and developer activities to consider, including their bug resolving history on GitHub or their activities on the platform such as StackOverflow.

7.7 Chapter Summary

To enhance the automatic identification of experts across various software project repositories on GitHub, this chapter introduces a novel approach for representing developers' programming knowledge based on the Syntax Patterns (SPs) they have mastered. We evaluate their syntactic proficiency by analyzing the distribution of these SPs in their commits. Our analysis reveals that this distribution adheres to Zipf's law, and the parameters of this distribution reflect the developers' syntactic mastery. To assess the effectiveness of our proposed model in identifying programming experts, we conducted a comparative analysis of our model's performance against that of different state-of-the-art methods. The results demonstrate that our proposed model surpasses the existing approaches in the binary classification of programming experts.

CHAPTER 8 CONCLUSION

In this chapter, we summarize our findings and conclude the thesis. In addition, we discuss the limitations of our proposed approaches and outline some directions for future work.

8.1 Summary of Works

The need for intelligent tools to assist all individuals involved in SE projects, including developers, is continuously growing. Beyond the advantages of these tools in improving developers' productivity, their impact on improving software quality is significant. The effectiveness of assistant tools influences the trust that developers place in them and their adoption in various tasks. This thesis explores how to elevate and enhance the correctness and effectiveness of intelligent assistant tools designed for various SE tasks. Specifically, the thesis focuses on three specific SE tasks: code generation, test generation, and expert identification. Below, we provide a summary of the proposed approaches and the key findings discussed throughout this thesis.

Empirical study on a state-of-the-art intelligent programming assistant tool, i.e., Github Copilot — To identify the strengths and limitations of programming intelligent assistant in code generation tasks, we conduct an empirical study comparing the quality of code generated by Copilot and code written by humans. Our results indicate that Copilot generates more buggy code in comparison to novice developers. However, the cost of repairing Copilot's buggy code is lower compared to human buggy code. We employ an APR tool to repair the buggy code of Copilot to increase its reliability. These findings led us to introduce a taxonomy of bugs observed in code generated by different LLMs. This taxonomy shows new categories of bugs that are distinct from bugs created by humans. Furthermore, our study reveals that Copilot generates code with lower complexity than that generated by novice human developers. However, our results also highlight the presence of numerous duplicates in the code generated by Copilot for a single programming task, despite Copilot's claims of removing duplicate solutions in its top 10 suggestions. Many of these duplicate codes only differ in variable or function names (natural language content), a distinction that Copilot does not consider when comparing sequences of tokens to identify duplicates.

Enhancing the effectiveness of test cases generated by LLMs — We endeavor to leverage the capabilities of LLMs for generating test cases by proposing a method named MuTAP. We first evaluate the effectiveness of test cases generated by LLMs compared to those

produced by a state-of-the-art tool, Pynguin. This assessment encompasses both artificial bugs (mutants) and real buggy code and highlights their limitation in revealing bugs. Furthermore, we propose a method to further enhance the effectiveness of test cases generated by LLMs. This enhancement involves augmenting the initial prompt with surviving mutants. While surviving mutants highlight the weaknesses in unit tests, none of the previous studies on assistant tools for test case generation leveraged the advantage of the surviving mutants to improve the effectiveness of test cases in detecting bugs. The results of our study, much like the first study, underscore the significance of post-processing steps in increasing the reliability and effectiveness of code (in this case, test cases) generated by LLMs.

Representing the domain expertise of developers — We also introduce dev2vec, a method for representing the domain expertise of developers based on their contributions across different projects on GitHub. This method leverages the capability of the doc2vec technique. Our study marks an initial step towards the long-standing dream in SE to identify experts for a task/issue and to select candidates for a project. We evaluate the effectiveness of dev2vec in representing developers' expertise in a common issue in SE: assessing the technical specialization of developers. The results show that dev2vec outperforms various baselines and state-of-the-art methods in classifying developers in 5 different job roles.

Assessing the expertise of developers — In this part, we propose a theoretical approach to modeling developer expertise and identifying experts. We draw inspiration from a theoretical concept in linguistics known as Zipf's law and apply it to represent the distribution of programming syntax patterns in code authored by different developers. Our findings demonstrate that the distribution of programming syntax patterns in developers' code serves as an indicator of their expertise. In addition, the parameter alpha in Zipf's law, combined with quantitative metrics of developers' activities, such as the number of commits, surpasses state-of-the-art studies in identifying expert developers in Python programming.

8.2 Limitations

- While a notable advantage of our proposed methods, such as MuTAP, is the ability to leverage other LLMs in the future, it is important to recognize that the limitations and performance of these models may also have an impact on our results. In addition to the model's performance, another relevant factor in the correctness of code generated by LLMs is the complexity or ambiguity of the prompt. There is a lack of datasets featuring real programming task specifications as prompts along with their corresponding reference code or solutions. In this thesis, our focus was not on the difficulty level of programming tasks that LLMs can handle, but rather on the quality of the code

and effectiveness of tests that they generate. Due to the existing capabilities of LLMs and available datasets, our choice of the dataset for programming tasks was limited to available datasets for which LLMs could generate code based on the provided task specifications. As a result, we conducted further investigation on quality and effectiveness assessment, but it is important to note that the datasets that we employed in our studies may not fully represent the complexity of real-world programming tasks. It is worth noting that our studies can be extended to encompass more complex programming tasks as future studies.

- While in our empirical assessments on the intelligent programming assistant tool, Copilot, we employed a variety of quantitative metrics, drawing inspiration from prior research in SE [17, 79, 182], the chosen metrics primarily focus on functional suitability and maintainability. However, in line with ISO/IEC 25010 and related studies [243], there are other quality characteristics to consider, including usability of software systems, performance efficiency, and security concerns [244]. We have deferred the assessment of these characteristics to future studies.

- In MuTAP, our primary focus was on generating tests for a PUT. MuTAP, like some other well-known test case generation assistant tools [23, 111], operates under the assumption that the PUT is not buggy. However, this assumption can introduce limitations in generating test cases when the PUT itself is also buggy.

- The adaptability of test cases to software changes is another important factor in test case generation. While MuTAP generates test cases for the presented PUT, its current version does not accommodate future changes to the PUT. When the PUT undergoes modifications, new test cases may need to be added, and some test cases may need to be removed to align the test suites with the updated PUT. While the current version of MuTAP does not emphasize test adaptation, it can be easily extended to accommodate new changes in the PUT, without re-designing.

- One of the limitations in studies that focus on building intelligent assistant tools to assess the expertise of developers in SE is the absence of data with ground truth that encompasses various developers with varying levels and domains of expertise. This gap is primarily due to the challenging nature of evaluating expertise, which typically necessitates a team of experts to assess developers' levels and domains of expertise. Additionally, previous studies have indicated that self-assessment by developers often results in underestimation or overestimation of levels [15]. However, to develop more accurate intelligent assistant tools for assessing developers' expertise across multiple

projects, a more comprehensive dataset with ground truth is essential. We believe that the limited studies in creating intelligent assistant tools for assessing and representing developers' expertise across different projects can be attributed to the absence of such datasets.

- Most of our studies and investigations in this thesis focus on the Python programming language. However, our approaches can be applied to other programming languages since they are not language-specific. But, generalizing them to other programming languages, such as Java, may encounter some new challenges that are not considered in this thesis.

8.3 Future Research

In addition to future studies that can aim at addressing the limitations of this thesis, several other compelling research avenues hold promise for developing more precise and effective intelligent assistant tools that enhance developers' productivity and contribute to the creation of high-quality software. In the future, researchers could extend our study in the following directions:

- **Automated code repair with LLMs:** Future studies can focus on Leveraging LLMs to effectively repair buggy code through a comprehensive methodology, including evolving bug localization, subsequent repair, and proper prompt engineering, reducing the time developers spend on debugging and verifying the output of LLMs.
- **Test case generation with LLMs:** While this thesis outlined techniques to build LLM-based intelligent assistant tools to generate test cases that rely on a correct PUT, there is potential in exploring more approaches to generate test cases using LLMs for code snippets, even in the presence of bugs in PUT. This could lead to more adaptive testing tools.
- **Enhancing Mutation Testing (MT) with LLMs:** Inspired by the taxonomy of LLM bugs presented in Chapter 4, future studies can contribute to the enhancement of assistant tools for conducting MT. Previous MT tools typically derive their mutation operators from the bugs observed in human-written buggy code. In future studies, incorporating new mutant operators inspired by the taxonomy of LLM bugs can adapt existing MT tools to reveal bugs specific to LLM-generated code. This direction holds promise for improving the quality and effectiveness of MT with LLMs.

- **Detecting LLM-generated code:** Another avenue for future research is the extraction of rules from the LLM bug taxonomy to build techniques that can effectively detect LLM-generated code based on bug patterns observed in them. This area has gained significant attention recently and holds potential for advancing code quality assessment.
- **Impact of intelligent programming assistants on developer expertise:** Investigating the impact of new AI programming assistant tools, such as Copilot, on the expertise of developers is a valuable research direction. Exploring how these tools correlate with users' expertise levels, impact their skills like code style and whether these tools enhance the quality of code written by developers with varying levels of expertise can provide valuable insights.
- **Skill transfer and knowledge expansion with intelligent Programming assistant tools:** Future studies can delve into whether intelligent programming assistant tools can help transfer novice developers to experts over time and expand their knowledge space. Additionally, examining whether developers incorporate new programming patterns, keywords, and built-in functions suggested by these assistant tools into their expertise or continue to rely on these tools for various programming tasks is an intriguing research area.
- **LLMs in other SE tasks:** Exploring the capabilities of LLMs in building intelligent assistant tools for other SE tasks, such as refactoring, offers a promising avenue for future investigations. Understanding how LLMs can enhance various aspects of software development beyond code generation and testing can be also an interesting area of research.

REFERENCES

[1] Y. H. et al., "Refactory: Re-factoring based program repair applied to programming assignments," GitHub, 2019. [Online]. Available: https://github.com/githubhuyang/refactory

[2] A. Arcuri, "An experience report on applying software testing academic results in industry: we need usable automated test generation," *Empirical Software Engineering*, vol. 23, pp. 1959–1981, 2018.

[3] J. T. Liang, C. Yang, and B. A. Myers, "A large-scale survey on the usability of ai programming assistants: Successes and challenges," in *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2023, pp. 605–617.

[4] A. M. Eilertsen and G. C. Murphy, "The usability (or not) of refactoring tools," in *2021 IEEE international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 2021, pp. 237–248.

[5] A. E. Hassan, "The road ahead for mining software repositories," in *2008 frontiers of software maintenance*. IEEE, 2008, pp. 48–57.

[6] R. Jain and U. Suman, "A systematic literature review on global software development life cycle," *ACM SIGSOFT Software Engineering Notes*, vol. 40, no. 2, pp. 1–14, 2015.

[7] J. T. Liang, C. Yang, and B. A. Myers, "Understanding the usability of ai programming assistants," *arXiv preprint arXiv:2303.17125*, 2023.

[8] D. Wong, A. Kothig, and P. Lam, "Exploring the verifiability of code generated by github copilot," *arXiv preprint arXiv:2209.01766*, 2022.

[9] B. Boehm, "A view of 20th and 21st century software engineering," in *Proceedings of the 28th international conference on Software engineering*, 2006, pp. 12–29.

[10] V. Garousi and J. Zhi, "A survey of software testing practices in canada," *Journal of Systems and Software*, vol. 86, no. 5, pp. 1354–1376, 2013.

[11] M. S. Bressler, "Building the winning organization through high-impact hiring," *Journal of Management and Marketing Research*, vol. 15, p. 1, 2014.

[12] B. Curtis, H. Krasner, and N. Iscoe, "A field study of the software design process for large systems," *Communications of the ACM*, vol. 31, no. 11, pp. 1268–1287, 1988.

[13] T. DeMarco and T. Lister, *Peopleware: productive projects and teams*. Addison-Wesley, 2013.

[14] D. Matter, A. Kuhn, and O. Nierstrasz, "Assigning bug reports using a vocabulary-based expertise model of developers," in *2009 6th IEEE international working conference on mining software repositories*. IEEE, 2009, pp. 131–140.

[15] J. E. Montandon, L. L. Silva, and M. T. Valente, "Identifying experts in software libraries and frameworks among github users," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 276–287.

[16] F. Zhang *et al.*, "An empirical study on factors impacting bug fixing time," in *2012 19th Working conference on reverse engineering*. IEEE, 2012, pp. 225–234.

[17] N. Nguyen and S. Nadi, "An empirical evaluation of GitHub Copilot's code suggestions," in *Accepted for publication Proceedings of the 19th ACM International Conference on Mining Software Repositories (MSR)*, 2022, pp. 1–5.

[18] I. Drori and N. Verma, "Solving linear algebra by program synthesis," *arXiv preprint arXiv:2111.08171*, 2021.

[19] J. Finnie-Ansley *et al.*, "The robots are coming: Exploring the implications of openai codex on introductory programming," in *Australasian Computing Education Conference*, 2022, pp. 10–19.

[20] A. Panichella *et al.*, "Revisiting test smells in automatically generated tests: limitations, pitfalls, and opportunities," in *2020 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 2020, pp. 523–533.

[21] F. Palomba *et al.*, "On the diffusion of test smells in automatically generated test code: An empirical study," in *Proceedings of the 9th international workshop on search-based software testing*, 2016, pp. 5–14.

[22] C. Lemieux *et al.*, "Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models," in *Accepted by 45th International Conference on Software Engineering (ICSE)*, 2023.

[23] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 416–419.

[24] D. Serra *et al.*, "On the effectiveness of manual and automatic unit test generation: ten years later," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 121–125.

[25] D. W. McDonald and M. S. Ackerman, "Expertise recommender: a flexible recommendation system and architecture," in *Proceedings of the 2000 ACM conference on Computer supported cooperative work*, 2000, pp. 231–240.

[26] A. Mockus and J. D. Herbsleb, "Expertise browser: a quantitative approach to identifying expertise," in *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*. IEEE, 2002, pp. 503–512.

[27] J. Anvik and G. C. Murphy, "Determining implementation expertise from bug reports," in *Fourth International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007)*. IEEE, 2007, pp. 2–2.

[28] D. Schuler and T. Zimmermann, "Mining usage expertise from version archives," in *Proceedings of the 2008 international working conference on Mining software repositories*, 2008, pp. 121–124.

[29] J. Oliveira, M. Viggiato, and E. Figueiredo, "How well do you know this library? mining experts from source code analysis," in *Proceedings of the XVIII Brazilian Symposium on Software Quality*, 2019, pp. 49–58.

[30] J. E. Montandon, M. T. Valente, and L. L. Silva, "Mining the technical roles of github users," *Information and Software Technology*, vol. 131, p. 106485, 2021.

[31] M. Verdi *et al.*, "An empirical study of c++ vulnerabilities in crowd-sourced code examples," *IEEE Transactions on Software Engineering*, 2020.

[32] S. L. Vadlamani and O. Baysal, "Studying software developer expertise and contributions in stack overflow and github," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 312–323.

[33] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *International conference on machine learning*. PMLR, 2014, pp. 1188–1196.

[34] G. K. Zipf, "Human behaviour and the principle of least-effort. cambridge ma edn," *Reading: Addison-Wesley*, 1949.

[35] B. Min *et al.*, "Recent advances in natural language processing via large pre-trained language models: A survey," *ACM Computing Surveys*, vol. 56, no. 2, pp. 1–40, 2023.

[36] Y. Chang *et al.*, "A survey on evaluation of large language models," *arXiv preprint arXiv:2307.03109*, 2023.

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

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

[39] T. Brown *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[40] A. Chowdhery *et al.*, "Palm: Scaling language modeling with pathways," *arXiv preprint arXiv:2204.02311*, 2022.

[41] H. Touvron *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

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

[43] Q. Dong *et al.*, "A survey for in-context learning," *arXiv preprint arXiv:2301.00234*, 2022.

[44] P. H. Winston, "Learning and reasoning by analogy," *Communications of the ACM*, vol. 23, no. 12, pp. 689–703, 1980.

[45] T. Sun *et al.*, "Black-box tuning for language-model-as-a-service," in *International Conference on Machine Learning*. PMLR, 2022, pp. 20 841–20 855.

[46] J. Wei *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in Neural Information Processing Systems*, vol. 35, pp. 24 824–24 837, 2022.

[47] X. Jiang *et al.*, "Self-planning code generation with large language model," *arXiv preprint arXiv:2303.06689*, 2023.

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

[49] H. Husain *et al.*, "Codesearchnet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.

[50] A. K. Turzo *et al.*, "Towards automated classification of code review feedback to support analytics," *arXiv preprint arXiv:2307.03852*, 2023.

[51] A. Akli *et al.*, "Flakycat: Predicting flaky tests categories using few-shot learning," in *2023 IEEE/ACM International Conference on Automation of Software Test (AST)*. IEEE, 2023, pp. 140–151.

[52] S. Arshad, S. Abid, and S. Shamail, "Codebert for code clone detection: A replication study," in *2022 IEEE 16th International Workshop on Software Clones (IWSC)*. IEEE, 2022, pp. 39–45.

[53] Y. Chai *et al.*, "Cross-domain deep code search with meta learning," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 487–498.

[54] X. Zhou, D. Han, and D. Lo, "Assessing generalizability of codebert," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021, pp. 425–436.

[55] E. Dinella *et al.*, "Toga: a neural method for test oracle generation," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2130–2141.

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

[57] N. Nguyen and S. Nadi, "An empirical evaluation of github copilot's code suggestions," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 1–5.

[58] Y. Wang *et al.*, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *arXiv preprint arXiv:2109.00859*, 2021.

[59] C. Wang *et al.*, "No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 382–394.

[60] T. Ahmed and P. Devanbu, "Few-shot training LLMs for project-specific code-summarization," *arXiv preprint arXiv:2207.04237*, 2022.

[61] T. Yu, X. Gu, and B. Shen, "Code question answering via task-adaptive sequence-to-sequence pre-training," in *2022 29th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2022, pp. 229–238.

[62] S. Lu *et al.*, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *arXiv preprint arXiv:2102.04664*, 2021.

[63] G. K. Zipf, *Human behavior and the principle of least effort: An introduction to human ecology*. Ravenio Books, 2016.

[64] A. Mehri and S. M. Lashkari, "Power-law regularities in human language," *The European Physical Journal B*, vol. 89, no. 11, p. 241, 2016.

[65] H. Zhang, "Discovering power laws in computer programs," *Information processing & management*, vol. 45, no. 4, pp. 477–483, 2009.

[66] A. Clauset, C. R. Shalizi, and M. E. Newman, "Power-law distributions in empirical data," *SIAM review*, vol. 51, no. 4, pp. 661–703, 2009.

[67] R. Pustet, "Zipf and his heirs," *Language Sciences*, vol. 26, no. 1, pp. 1–25, 2004.

[68] W. Oh and H. Oh, "Pyter: effective program repair for python type errors," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 922–934.

[69] Y. Hu *et al.*, "Re-factoring based program repair applied to programming assignments," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 388–398.

[70] S. Gulwani, I. Radiček, and F. Zuleger, "Automated clustering and program repair for introductory programming assignments," *ACM SIGPLAN Notices*, vol. 53, no. 4, pp. 465–480, 2018.

[71] J. Cao *et al.*, "A study on prompt design, advantages and limitations of chatgpt for deep learning program repair," *arXiv preprint arXiv:2304.08191*, 2023.

[72] J. Zhang *et al.*, "Repairing bugs in python assignments using large language models," *arXiv preprint arXiv:2209.14876*, 2022.

[73] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Lexical statistical machine translation for language migration," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 651–654.

[74] S. Ren *et al.*, "Codebleu: a method for automatic evaluation of code synthesis," *arXiv preprint arXiv:2009.10297*, 2020.

[75] N. Tran *et al.*, "Does bleu score work for code migration?" in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 2019, pp. 165–176.

[76] C. Ebert *et al.*, "Cyclomatic complexity," *IEEE software*, vol. 33, no. 6, pp. 27–29, 2016.

[77] M. M. S. Sarwar, S. Shahzad, and I. Ahmad, "Cyclomatic complexity: The nesting problem," in *Eighth International Conference on Digital Information Management (ICDIM 2013)*. IEEE, 2013, pp. 274–279.

[78] S. Scalabrino *et al.*, "Automatically assessing code understandability," *IEEE Transactions on Software Engineering*, vol. 47, no. 3, pp. 595–613, 2019.

[79] S. Fakhoury *et al.*, "Improving source code readability: Theory and practice," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 2019, pp. 2–12.

[80] C. E. C. Dantas and M. A. Maia, "Readability and understandability scores for snippet assessment: an exploratory study," *arXiv preprint arXiv:2108.09181*, 2021.

[81] R. Alur *et al.*, *Syntax-guided synthesis*. IEEE, 2013.

[82] F. Chauvel and J.-M. Jézéquel, "Code generation from uml models with semantic variation points," in *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2005, pp. 54–68.

[83] C. Green, "Application of theorem proving to problem solving," in *Readings in Artificial Intelligence*. Elsevier, 1981, pp. 202–222.

[84] S. Gulwani, "Automating string processing in spreadsheets using input-output examples," *ACM Sigplan Notices*, vol. 46, no. 1, pp. 317–330, 2011.

[85] S. Imai, "Is github copilot a substitute for human pair-programming? an empirical study," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, 2022, pp. 319–321.

[86] F. A. Sakib, S. H. Khan, and A. Karim, "Extending the frontier of chatgpt: Code generation and debugging," *arXiv preprint arXiv:2307.08260*, 2023.

[87] D. Sobania, M. Briesch, and F. Rothlauf, "Choose your programming copilot: A comparison of the program synthesis performance of github copilot and genetic programming," *arXiv preprint arXiv:2111.07875*, 2021.

[88] P. Vaithilingam, T. Zhang, and E. L. Glassman, "Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models," in *CHI Conference on Human Factors in Computing Systems Extended Abstracts*, 2022, pp. 1–7.

[89] L. Tang *et al.*, "Solving probability and statistics problems by program synthesis," *arXiv preprint arXiv:2111.08267*, 2021.

[90] H. Pearce *et al.*, "Asleep at the keyboard? assessing the security of github copilot's code contributions," in *2022 2022 IEEE Symposium on Security and Privacy (SP) (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2022, pp. 980–994. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP46214.2022.00057

[91] O. Asare, M. Nagappan, and N. Asokan, "Is github's copilot as bad as humans at introducing vulnerabilities in code?" *arXiv preprint arXiv:2204.04741*, 2022.

[92] E. A. Moroz, V. O. Grizkevich, and I. M. Novozhilov, "The potential of artificial intelligence as a method of software developer's productivity improvement," in *2022 Conference of Russian Young Researchers in Electrical and Electronic Engineering (ElConRus)*. IEEE, 2022, pp. 386–390.

[93] A. Ziegler *et al.*, "Productivity assessment of neural code completion," *arXiv preprint arXiv:2205.06537*, 2022.

[94] N. Forsgren *et al.*, "The space of developer productivity: There's more to it than you think." *Queue*, vol. 19, no. 1, pp. 20–48, 2021.

[95] B. A. Becker *et al.*, "Programming is hard–or at least it used to be: Educational opportunities and challenges of ai code generation," *arXiv preprint arXiv:2212.01020*, 2022.

[96] P. Denny, V. Kumar, and N. Giacaman, "Conversing with copilot: Exploring prompt engineering for solving cs1 problems using natural language," in *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, 2023, pp. 1136–1142.

[97] M. Wermelinger, "Using github copilot to solve simple programming problems," in *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, ser. SIGCSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 172–178. [Online]. Available: https://doi.org/10.1145/3545945.3569830

[98] J. Leinonen *et al.*, "Using large language models to enhance programming error messages," in *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, ser. SIGCSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 563–569. [Online]. Available: https://doi.org/10.1145/3545945.3569770

[99] J. Zamfirescu-Pereira *et al.*, "Why johnny can't prompt: how non-ai experts try (and fail) to design llm prompts," in *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, 2023, pp. 1–21.

[100] E. Jones and J. Steinhardt, "Capturing failures of large language models via human cognitive biases," *Advances in Neural Information Processing Systems*, vol. 35, pp. 11 785–11 799, 2022.

[101] C. Bird *et al.*, "Taking flight with copilot: Early insights and opportunities of ai-powered pair-programming tools," *Queue*, vol. 20, no. 6, pp. 35–57, 2022.

[102] S. Barke, M. B. James, and N. Polikarpova, "Grounded copilot: How programmers interact with code-generating models," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA1, pp. 85–111, 2023.

[103] J. Shore and S. Warden, *The art of agile development*, 2nd ed. "O'Reilly", 2021.

[104] S. Siddiqui, *Learning Test-Driven Development*. "O'Reilly", 2021.

[105] T. Xie, "Augmenting automatically generated unit-test suites with regression oracle checking," in *ECOOP 2006–Object-Oriented Programming: 20th European Conference, Nantes, France, July 3-7, 2006. Proceedings 20*. Springer, 2006, pp. 380–403.

[106] M. Selakovic *et al.*, "Test generation for higher-order functions in dynamic languages," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–27, 2018.

[107] E. Arteca *et al.*, "Nessie: automatically testing javascript apis with asynchronous callbacks," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1494–1505.

[108] K. Sen, D. Marinov, and G. Agha, "Cute: A concolic unit testing engine for c," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 263–272, 2005.

[109] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005, pp. 213–223.

[110] G. Fraser and A. Arcuri, "Evolutionary generation of whole test suites," in *2011 11th International Conference on Quality Software*. IEEE, 2011, pp. 31–40.

[111] S. Lukasczyk and G. Fraser, "Pynguin: Automated unit test generation for python," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, 2022, pp. 168–172.

[112] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2012.

[113] P. Bareiß *et al.*, "Code generation tools (almost) for free? a study of few-shot, pre-trained language models on code," *arXiv preprint arXiv:2206.01335*, 2022.

[114] M. Schäfer *et al.*, "Adaptive test generation using a large language model," *arXiv preprint arXiv:2302.06527*, 2023.

[115] S. Kang, J. Yoon, and S. Yoo, "Large language models are few-shot testers: Exploring LLM-based general bug reproduction," *arXiv preprint arXiv:2209.11515*, 2022.

[116] N. Nashid, M. Sintaha, and A. Mesbah, "Retrieval-based prompt selection for code-related few-shot learning," 2023.

[117] M. Tufano *et al.*, "Unit test case generation with transformers and focal context," *arXiv preprint arXiv:2009.05617*, 2021.

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

[119] S. K. Lahiri *et al.*, "Interactive code generation via test-driven user-intent formalization," *arXiv preprint arXiv:2208.05950*, 2022.

[120] B. Chen *et al.*, "Codet: Code generation with generated tests," *arXiv preprint arXiv:2207.10397*, 2022.

[121] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023). Association for Computing Machinery*, 2023.

[122] N. Jiang *et al.*, "Impact of code language models on automated program repair," *arXiv preprint arXiv:2302.05020*, 2023.

[123] J. A. Prenner, H. Babii, and R. Robbes, "Can openai's codex fix bugs? an evaluation on quixbugs," in *Proceedings of the Third International Workshop on Automated Program Repair*, 2022, pp. 69–75.

[124] D. Sobania *et al.*, "An analysis of the automatic bug fixing performance of chatgpt arxiv," *arXiv preprint arXiv:2301.08653*, 2023.

[125] H. Joshi *et al.*, "Repair is nearly generation: Multilingual program repair with llms," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 37, no. 4, 2023, pp. 5131–5140.

[126] N. Nashid, M. Sintaha, and A. Mesbah, "Retrieval-based prompt selection for code-related few-shot learning," in *Proceedings of the 45th International Conference on Software Engineering (ICSE'23)*, 2023.

[127] V. Liventsev *et al.*, "Fully autonomous programming with large language models," *arXiv preprint arXiv:2304.10423*, 2023.

[128] S. Minto and G. C. Murphy, "Recommending emergent teams," in *Fourth International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007)*. IEEE, 2007, pp. 5–5.

[129] Y. Tian *et al.*, "Learning to rank for bug report assignee recommendation," in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. IEEE, 2016, pp. 1–10.

[130] D. Kim *et al.*, "Where should we fix this bug? a two-phase recommendation model," *IEEE transactions on software Engineering*, vol. 39, no. 11, pp. 1597–1610, 2013.

[131] J. Zhang, M. S. Ackerman, and L. Adamic, "Expertise networks in online communities: structure and algorithms," in *Proceedings of the 16th international conference on World Wide Web*, 2007, pp. 221–230.

[132] X. Cheng *et al.*, "Exploiting user feedback for expert finding in community question answering," in *2015 IEEE International Conference on Data Mining Workshop (ICDMW)*. IEEE, 2015, pp. 295–302.

[133] A. Pal, F. M. Harper, and J. A. Konstan, "Exploring question selection bias to identify experts and potential experts in community question answering," *ACM Transactions on Information Systems (TOIS)*, vol. 30, no. 2, pp. 1–28, 2012.

[134] N. Dalvi *et al.*, "Aggregating crowdsourced binary ratings," in *Proceedings of the 22nd international conference on World Wide Web*, 2013, pp. 285–294.

[135] A. S. Badashian, "Realistic bug triaging," in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2016, pp. 847–850.

[136] R. Sindhgatta, "Identifying domain expertise of developers from source code," in *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2008, pp. 981–989.

[137] A. S. Badashian, A. Hindle, and E. Stroulia, "Crowdsourced bug triaging: Leveraging q&a platforms for bug assignment," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2016, pp. 231–248.

[138] C. Teyton *et al.*, "Automatic extraction of developer expertise," in *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, 2014, pp. 1–10.

[139] R. Venkataramani *et al.*, "Discovery of technical expertise from open source code repositories," in *Proceedings of the 22nd International Conference on World Wide Web*, 2013, pp. 97–98.

[140] A. Sajedi-Badashian and E. Stroulia, "Vocabulary and time based bug-assignment: A recommender system for open-source projects," *Software: Practice and Experience*, 2020.

[141] P. L. Li, A. J. Ko, and A. Begel, "What distinguishes great software engineers?" *Empirical Software Engineering*, vol. 25, no. 1, pp. 322–352, 2020.

[142] X. Xia *et al.*, "Dual analysis for recommending developers to resolve bugs," *Journal of Software: Evolution and Process*, vol. 27, no. 3, pp. 195–220, 2015.

[143] M. Allahbakhsh *et al.*, "Quality control in crowdsourcing systems: Issues and directions," *IEEE Internet Computing*, vol. 17, no. 2, pp. 76–81, 2013.

[144] W. Huang *et al.*, "Cpdscorer: Modeling and evaluating developer programming ability across software communities." in *SEKE*, 2016, pp. 87–92.

[145] G. J. Greene and B. Fischer, "Cvexplorer: Identifying candidate developers by mining and exploring their open source contributions," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 804–809.

[146] G. Maskeri, S. Sarkar, and K. Heafield, "Mining business topics in source code using latent dirichlet allocation," in *Proceedings of the 1st India software engineering conference*, 2008, pp. 113–120.

[147] D. Binkley *et al.*, "Understanding lda in source code analysis," in *Proceedings of the 22nd international conference on program comprehension*, 2014, pp. 26–36.

[148] T. Savage *et al.*, "Topic xp: Exploring topics in source code using latent dirichlet allocation," in *2010 IEEE International Conference on Software Maintenance*. IEEE, 2010, pp. 1–6.

[149] A. Kuhn, S. Ducasse, and T. Gírba, "Semantic clustering: Identifying topics in source code," *Information and software technology*, vol. 49, no. 3, pp. 230–243, 2007.

[150] X. Xie *et al.*, "Dretom: Developer recommendation based on topic models for bug resolution," in *Proceedings of the 8th international conference on predictive models in software engineering*, 2012, pp. 19–28.

[151] P. Louridas, D. Spinellis, and V. Vlachos, "Power laws in software," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 18, no. 1, pp. 1–26, 2008.

[152] N. Walkinshaw and L. Minku, "Are 20% of files responsible for 80% of defects?" in *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2018, pp. 1–10.

[153] C. Casalnuovo, K. Sagae, and P. Devanbu, "Studying the difference between natural and programming language corpora," *Empirical Software Engineering*, vol. 24, no. 4, pp. 1823–1868, 2019.

[154] E. Shulzinger, I. Legchenkova, and E. Bormashenko, "Co-occurrence of the benford-like and zipf laws arising from the texts representing human and artificial languages," *arXiv preprint arXiv:1803.03667*, 2018.

[155] M. L. Goldstein, S. A. Morris, and G. G. Yen, "Problems with fitting to the power-law distribution," *The European Physical Journal B-Condensed Matter and Complex Systems*, vol. 41, no. 2, pp. 255–258, 2004.

[156] H. Bauke, "Parameter estimation for power-law distributions by maximum likelihood methods," *The European Physical Journal B*, vol. 58, no. 2, pp. 167–173, 2007.

[157] A. Deluca and Á. Corral, "Fitting and goodness-of-fit test of non-truncated and truncated power-law distributions," *Acta Geophysica*, vol. 61, no. 6, pp. 1351–1394, 2013.

[158] E. G. Altmann and M. Gerlach, "Statistical laws in linguistics," in *Creativity and universality in language*. Springer, 2016, pp. 7–26.

[159] J. Baixeries, B. Elvevåg, and R. Ferrer-i Cancho, "The evolution of the exponent of zipf's law in language ontogeny," *PloS one*, vol. 8, no. 3, p. e53227, 2013.

[160] Y. Zhang *et al.*, "ilinker: a novel approach for issue knowledge acquisition in github projects," *World Wide Web*, vol. 23, no. 3, pp. 1589–1619, 2020.

[161] U. Alon *et al.*, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[162] R. C. Lozoya *et al.*, "Commit2vec: Learning distributed representations of code changes," *SN Computer Science*, vol. 2, no. 3, pp. 1–16, 2021.

[163] B. Theeten, F. Vandeputte, and T. Van Cutsem, "Import2vec: Learning embeddings for software libraries," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 18–28.

[164] T. Dey, A. Karnauch, and A. Mockus, "Representation of developer expertise in open source software," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 995–1007.

[165] Y. H. et al., "Refactory," program-repair.org, 2019. [Online]. Available: https://program-repair.org/benchmarks.html

[166] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *SIGPLAN Not.*, vol. 39, no. 12, p. 92–106, dec 2004. [Online]. Available: https://doi.org/10.1145/1052883.1052895

[167] D. Hovemeyer, J. Spacco, and W. Pugh, "Evaluating and tuning a static analysis to find null pointer bugs," in *Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, 2005, pp. 13–19.

[168] N. Carlini *et al.*, "Quantifying memorization across neural language models," *arXiv preprint arXiv:2202.07646*, 2022.

[169] U. Z. Ahmed *et al.*, "Characterizing the pedagogical benefits of adaptive feedback for compilation errors by novice programmers," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering Education and Training*, 2020, pp. 139–150.

[170] H. Tian *et al.*, "Is chatgpt the ultimate programming assistant–how far is it?" *arXiv preprint arXiv:2304.11938*, 2023.

[171] J. Heo *et al.*, "Referent: Transformer-based feedback generation using assignment information for programming course," in *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*. IEEE, 2023, pp. 101–106.

[172] Y. Peng *et al.*, "Domain knowledge matters: Improving prompts with fix templates for repairing python type errors," *arXiv preprint arXiv:2306.01394*, 2023.

[173] C. Koutcheme *et al.*, "Automated program repair using generative models for code infilling," in *International Conference on Artificial Intelligence in Education*. Springer, 2023, pp. 798–803.

[174] C. Koutcheme, "Training language models for programming feedback using automated repair tools," in *International Conference on Artificial Intelligence in Education*. Springer, 2023, pp. 830–835.

[175] H. Yu *et al.*, "Codereval: A benchmark of pragmatic code generation with generative pre-trained models," *arXiv preprint arXiv:2302.00288*, 2023.

[176] E. Nijkamp *et al.*, "Codegen: An open large language model for code with multi-turn program synthesis," *arXiv preprint arXiv:2203.13474*, 2022.

[177] F. Christopoulou *et al.*, "Pangu-coder: Program synthesis with function-level language modeling," *arXiv preprint arXiv:2207.11280*, 2022.

[178] X. Chen *et al.*, "Teaching large language models to self-debug," *arXiv preprint arXiv:2304.05128*, 2023.

[179] D. Zan *et al.*, "When neural model meets nl2code: A survey," *arXiv preprint arXiv:2212.09420*, 2022.

[180] J. Bogatinovski and O. Kao, "Auto-logging: Ai-centred logging instrumentation," in *2023 IEEE/ACM 45th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE, 2023, pp. 95–100.

[181] Y. Li *et al.*, "Competition-level code generation with alphacode," *arXiv preprint arXiv:2203.07814*, 2022.

[182] S. Kim and E. J. Whitehead Jr, "How long did it take to fix bugs?" in *Proceedings of the 2006 international workshop on Mining software repositories*, 2006, pp. 173–174.

[183] K. Papineni *et al.*, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.

[184] A. Arcuri, "On the automation of fixing software bugs," in *Companion of the 30th international conference on Software engineering*, 2008, pp. 1003–1006.

[185] P. Salazar Paredes *et al.*, "Comparing Python programs using abstract syntax trees," Uniandes, Tech. Rep., 2020.

[186] L. M. Maruping, X. Zhang, and V. Venkatesh, "Role of collective ownership and coding standards in coordinating expertise in software project teams," *European Journal of Information Systems*, vol. 18, no. 4, pp. 355–371, 2009.

[187] R. M. dos Santos and M. A. Gerosa, "Impacts of coding practices on readability," in *Proceedings of the 26th Conference on Program Comprehension*, 2018, pp. 277–285.

[188] L. Plonka *et al.*, "Knowledge transfer in pair programming: An in-depth analysis," *International journal of human-computer studies*, vol. 73, pp. 66–78, 2015.

[189] K. M. Lui and K. C. Chan, "Pair programming productivity: Novice–novice vs. expert–expert," *International Journal of Human-computer studies*, vol. 64, no. 9, pp. 915–925, 2006.

[190] I. Fronza, A. Sillitti, and G. Succi, "An interpretation of the results of the analysis of pair programming during novices integration in a team," in *2009 3rd International Symposium on Empirical Software Engineering and Measurement*. IEEE, 2009, pp. 225–235.

[191] C. B. Seaman, "Qualitative methods in empirical studies of software engineering," *IEEE Transactions on software engineering*, vol. 25, no. 4, pp. 557–572, 1999.

[192] N. Humbatova *et al.*, "Taxonomy of real faults in deep learning systems," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1110–1121.

[193] M. X. Liu *et al.*, ""what it wants me to say": Bridging the abstraction gap between end-user programmers and code-generating large language models," in *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, 2023, pp. 1–31.

[194] C. Wohlin *et al.*, *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[195] M. Tufano *et al.*, "Generating accurate assert statements for unit test cases using pre-trained transformers," in *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test*, 2022, pp. 54–64.

[196] X. Cai and M. R. Lyu, "The effect of code coverage on fault detection under different testing profiles," in *Proceedings of the 1st International Workshop on Advances in Model-Based Testing*, ser. A-MOST '05. New York, NY, USA: ACM, 2005, p. 1–7. [Online]. Available: https://doi.org/10.1145/1083274.1083288

[197] R. Gopinath, C. Jensen, and A. Groce, "Code coverage for suite evaluation by developers," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 72–82.

[198] H. Hemmati, "How effective are code coverage criteria?" in *2015 IEEE International Conference on Software Quality, Reliability and Security*. IEEE, 2015, pp. 151–156.

[199] S. Lukasczyk, F. Kroiß, and G. Fraser, "An empirical study of automated unit test generation for python," *Empirical Software Engineering*, vol. 28, no. 2, p. 36, 2023.

[200] A. Arcuri and G. Fraser, "Parameter tuning or default values? an empirical investigation in search-based software engineering," *Empirical Software Engineering*, vol. 18, pp. 594–623, 2013.

[201] P. Liu *et al.*, "Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing," *ACM Computing Surveys*, vol. 55, no. 9, pp. 1–35, 2023.

[202] H. Joshi *et al.*, "Repair is nearly generation: Multilingual program repair with LLMs," *arXiv preprint arXiv:2208.11640*, 2022.

[203] K. Hałas, "Mutpy: a mutation testing tool for Python 3.x source code," https://github.com/mutpy/mutpy, 2019.

[204] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," in *Proceedings of the 19th international symposium on Software testing and analysis*, 2010, pp. 147–158.

[205] A. Arcuri, "Test suite generation with the Many Independent Objective (MIO) algorithm," *Information and Software Technology*, vol. 104, pp. 195–206, 2018.

[206] A. e. a. Panichella, "Reformulating branch coverage as a many-objective optimization problem," in *2015 IEEE 8th international conference on software testing, verification and validation (ICST)*. IEEE, 2015, pp. 1–10.

[207] A. Panichella, F. M. Kifetew, and P. Tonella, "Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets," *IEEE Transactions on Software Engineering*, vol. 44, no. 2, pp. 122–158, 2017.

[208] D. Shrivastava, H. Larochelle, and D. Tarlow, "Repository-level prompt generation for large language models of code," *arXiv preprint arXiv:2206.12839*, 2022.

[209] I. Alvarado, I. Gazit, and A. Wattenberger, "Github copilot labs," https://githubnext.com/projects/copilot-labs/, 2023.

[210] H. Lightman *et al.*, "Let's verify step by step," *arXiv preprint arXiv:2305.20050*, 2023.

[211] M. Papadakis *et al.*, "Mutation testing advances: an analysis and survey," in *Advances in Computers*. Elsevier, 2019, vol. 112, pp. 275–378.

[212] M. L. Siddiq *et al.*, "Exploring the effectiveness of large language models in generating unit tests," *arXiv preprint arXiv:2305.00418*, 2023.

[213] D. Kim *et al.*, "Multi-co-training for document classification using various document representations: Tf–idf, lda, and doc2vec," *Information Sciences*, vol. 477, pp. 15–29, 2019.

[214] P. Leelaprute and S. Amasaki, "A comparative study on vectorization methods for non-functional requirements classification," *Information and Software Technology*, vol. 150, p. 106991, 2022.

[215] K. W. Church, "Word2vec," *Natural Language Engineering*, vol. 23, no. 1, pp. 155–162, 2017.

[216] W. Zhang *et al.*, "Finelocator: A novel approach to method-level fine-grained bug localization by query expansion," *Information and Software Technology*, vol. 110, pp. 121–135, 2019.

[217] L. Ge and T.-S. Moh, "Improving text classification with word embedding," in *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, 2017, pp. 1796–1805.

[218] Y. Wan *et al.*, "Scsminer: mining social coding sites for software developer recommendation with relevance propagation," *World Wide Web*, vol. 21, no. 6, pp. 1523–1543, 2018.

[219] T. Lucassen and J. M. Schraagen, "Factual accuracy and trust in information: The role of expertise," *Journal of the American Society for Information Science and Technology*, vol. 62, no. 7, pp. 1232–1242, 2011.

[220] J. Shlens, "A tutorial on principal component analysis," *arXiv preprint arXiv:1404.1100*, 2014.

[221] T. Z. Abdulhameed, I. Zitouni, and I. Abdel-Qader, "Enhancement of the word2vec class-based language modeling by optimizing the features vector using pca," in *2018 IEEE International Conference on Electro/Information Technology (EIT)*. IEEE, 2018, pp. 0866–0870.

[222] S. Liu *et al.*, "Visual exploration of semantic relationships in neural word embeddings," *IEEE transactions on visualization and computer graphics*, vol. 24, no. 1, pp. 553–562, 2017.

[223] Z.-Q. Wang *et al.*, "An optimal svm-based text classification algorithm," in *2006 International Conference on Machine Learning and Cybernetics*. IEEE, 2006, pp. 1378–1381.

[224] A. Bouaziz *et al.*, "Short text classification using semantic random forest," in *International Conference on Data Warehousing and Knowledge Discovery*. Springer, 2014, pp. 288–299.

[225] K. Shah *et al.*, "A comparative analysis of logistic regression, random forest and knn models for the text classification," *Augmented Human Research*, vol. 5, no. 1, pp. 1–16, 2020.

[226] A. Neishabouri and M. C. Desmarais, "Estimating the number of latent topics through a combination of methods," *Procedia Computer Science*, vol. 192, pp. 1190–1197, 2021.

[227] R. Wang *et al.*, "Convolutional recurrent neural networks for text classification," in *2019 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2019, pp. 1–6.

[228] Y. Zhang *et al.*, "Every document owns its structure: Inductive text classification via graph neural networks," *arXiv preprint arXiv:2004.13826*, 2020.

[229] J. Nam *et al.*, "Large-scale multi-label text classification—revisiting neural networks," in *Joint european conference on machine learning and knowledge discovery in databases*. Springer, 2014, pp. 437–452.

[230] J. Marlow and L. Dabbish, "Activity traces and signals in software developer recruitment and hiring," in *Proceedings of the 2013 conference on Computer supported cooperative work*, 2013, pp. 145–156.

[231] J. E. Montandon, M. T. Valente, and L. L. Silva, "Mining the technical roles of github users," *Information and Software Technology*, vol. 131, p. 106485, 2021.

[232] L. Singer *et al.*, "Mutual assessment in the social programmer ecosystem: An empirical investigation of developer profile aggregators," in *Proceedings of the 2013 conference on Computer supported cooperative work*, 2013, pp. 103–116.

[233] R. Edwards and L. Collins, "Lexical frequency profiles and zipf's law," *Language Learning*, vol. 61, no. 1, pp. 1–30, 2011.

[234] J.-C. Falmagne *et al.*, "Introduction to knowledge spaces: How to build, test, and search them." *Psychological Review*, vol. 97, no. 2, p. 201, 1990.

[235] B. Cui *et al.*, "Code comparison system based on abstract syntax tree," in *2010 3rd IEEE International Conference on Broadband Network and Multimedia Technology (ICBNMT)*. IEEE, 2010, pp. 668–673.

[236] L. Mou *et al.*, "Convolutional neural networks over tree structures for programming language processing," in *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[237] G. F. Jenks, "The data model concept in statistical mapping," *International yearbook of cartography*, vol. 7, pp. 186–190, 1967.

[238] R. McMaster, "In memoriam: George f. jenks (1916-1996)," *Cartography and Geographic Information Systems*, vol. 24, no. 1, pp. 56–59, 1997.

[239] Y. Boubekeur, G. Mussbacher, and S. McIntosh, "Automatic assessment of students' software models using a simple heuristic and machine learning," in *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, 2020, pp. 1–10.

[240] D. Spadini, M. Aniche, and A. Bacchelli, "Pydriller: Python framework for mining software repositories," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 908–911.

[241] T. Fritz *et al.*, "A degree-of-knowledge model to capture source code familiarity," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, 2010, pp. 385–394.

[242] J. R. da Silva *et al.*, "Niche vs. breadth: Calculating expertise over time through a fine-grained analysis," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015, pp. 409–418.

[243] N. Tsuda *et al.*, "Wsqf: Comprehensive software quality evaluation framework and benchmark based on square," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2019, pp. 312–321.

[244] T. Komiyama *et al.*, "Usability of software–intensive systems from developers' point of view: Current status and future perspectives of international standardization of usability evaluation," in *Human-Computer Interaction. Design and User Experience: Thematic Area, HCI 2020, Held as Part of the 22nd International Conference, HCII 2020, Copenhagen, Denmark, July 19–24, 2020, Proceedings, Part I 22*. Springer, 2020, pp. 450–463.

APPENDIX A TAXONOMY OF BUGS IN CODE GENERATED BY LLMS

As previously discussed in Chapter 4, we identified ten distinct bug categories within code generated by LLMs. This section will present an example for each category.

Misinterpretation

Misinterpretation occurs when the generated code deviates from the intended task prompt, often due to the model's struggle to comprehend the task specifications. In Listing A.1, you can find an example task along with its corresponding reference solution. The function declaration and the docstring at the beginning of the solution (lines 1 to 6) represent the prompt and the reset (lines 7 to 16) is the reference solution. In Listing A.2, we present a code generated by PanGu-Coder for the same prompt, which exhibits a misalignment with the given prompt and is labeled as *Misinterpretation*.

Listing A.1 Reference Solution for the task "int_to_string."

```
1  def int_to_string(number: int, alphabet: List[str], padding: Optional[int] = None) -> str:
2      """
3      Convert a number to a string, using the given alphabet.
4
5      The output has the most significant digit first.
6      """
7      output = ""
8      alpha_len = len(alphabet)
9      while number:
10         number, digit = divmod(number, alpha_len)
11         output += alphabet[digit]
12     if padding:
13         remainder = max(padding - len(output), 0)
14         output = output + alphabet[0] * remainder
15     return output[::-1]
```

Listing A.2 Code generated by PanGu-Coder and labeled as "Misinterpretation".

```
1  def int_to_string(number: int, alphabet: str) -> str:
2      return alphabet[number]
```

Syntax Error

Any syntax error, such as a missing parenthesis or semicolon, is categorized as a 'Syntax Error.' Listing A.3 showcases a code example with a syntax error in the first line generated by CodeGen for the task in Listing A.1.

Listing A.3 Code generated by CodeGen and labeled as "Syntax Error".

```
def int_to_string(number:
        """
        Convert a number to a string, using the given alphabet.

The output has the most significant digit first.
        """
        alphabet_string = ""
        for i in range(10):
                alphabet_string += chr(number % 10 + ord('A'))
                number //= 10

        return alphabet_string
```

Silly Mistake

We labeled issues such as redundant conditions or unnecessary casting as *Silly Mistake*. While these types of issues may not introduce a bug on their own, they have the potential to easily lead to one. In Listing A.4, there is an example generated by CodeGen where both the *if* and *else* conditions have the same statement. This is labeled as *Silly Mistake*.

Listing A.4 Code generated by CodeGen and labeled as "Silly Mistake".

```
if not logfile.exists():
                logging.basicConfig(
                        filename=str(logf),
                        format='%(asctime)s %(levelname)s %(name)s %(message)s',
                        datefmt='%Y-%m-%d %H:%M:%S',
                        level=logging.DEBUG,
                )
        else:
                logging.basicConfig(
                        filename=str(logf),
```

```
11                      format='%(asctime)s %(levelname)s %(name)s %(message)s',
12                      datefmt='%Y-%m-%d %H:%M:%S',
13                      level=logging.DEBUG,
14              )
```

Prompt-biased Code

LLM excessively relies on provided examples or words in the prompt while implementing a function and it hinders the generalization of the generated code. Listing A.5 illustrates this issue. Lines 1 to 9 constitute the prompt, while lines 10 to 13 showcase the code generated by Codex for this task. In this example, the generated code is accurate only for the specific example provided in the prompt, when the length of the vertex is 4.

Listing A.5 Code generated by Codex and labeled as "Prompt-biased Code".

```
1  def vertex3tuple(vertices):
2      """return 3 points for each vertex of the polygon. This will include the vertex and the 2 points
3      on both sides of the vertex:
4
5          polygon with vertices ABCD
6          Will return
7          DAB, ABC, BCD, CDA -> returns 3tuples
8          #A B C D -> of vertices
9      """
10     return [(vertices[-1], vertices[0], vertices[1]),
11                             (vertices[0], vertices[1], vertices[2]),
12                             (vertices[1], vertices[2], vertices[3]),
13                             (vertices[2], vertices[3], vertices[0])]
```

Missing Corner Case

This type occurs when the generated code operates correctly, except for overlooking certain corner cases. Listing A.6 provides an example of this type of bug, with lines 1 to 8 representing the prompt and line 9 showcasing the code generated by Codex. Although the Codex-generated code checks for various representations of localhost, the reference solution and test cases are expected to verify additional potential options, such as *socket.gethostname()*.

Listing A.6 Code generated by Codex and labeled as "Missing Corner Case".

```
import socket
def is_local(host):
    """
    Checks if the host is the localhost

    :param host: The hostname or ip
    :return: True if the host is the localhost
    """
    return host in ['localhost', '127.0.0.1', '::1']
```

Wrong Input Type

We use this label when LLM uses an incorrect input type in a correct function call. For example in Listing A.7, in the code generated by CodeGen, the python built-in function *min* is called on a ClassType input which raises an error.

Listing A.7 Code generated by CodeGen and labeled as "Wrong Input Type".

```
def minimalBases(classes):
    """Reduce a list of base classes to its ordered minimum equivalent"""
    def minimalBases(classes):
        """
        Reduce a list of base classes to its ordered minimum equivalent
        """
        if len(classes) == 1:
                return classes[0]
        else:
                return min(classes)
```

Function Hallucination

We apply this label when the LLM hallucinates by attempting to utilize a function that neither exists nor has been defined. For instance, in Listing A.8, within the code generated by Codex, it calls a function named "find_path_to_glob", which is not defined.

Listing A.8 Code generated by Codex and labeled as "Function Hallucination".

```
def make_find_paths(find_paths):
    '''
```

```
    Given a sequence of path fragments or patterns as passed to `--find`, transform all path
    fragments into glob patterns. Pass through existing patterns untouched.

    For example, given find_paths of:

      ['foo.txt', 'pp:root/somedir']

    ... transform that into:

      ['sh:**/*foo.txt*/**', 'pp:root/somedir']
    """
    return [find_path_to_glob(x) for x in find_paths]
```

Wrong Attribute

This label is applied when the LLM utilizes an incorrect attribute for an object or module, or when it hallucinates by attempting to use an attribute that does not exist. For example, in Listing A.9 from the code generated by PanGu-Coder, it tries to use 'flags' as an attribute of the input 'parser', which is not a correct attribute according to the information provided in the prompt.

Listing A.9 Code generated by PanGu-Coder and labeled as "Wrong Attribute".

```
def parser_flags(parser):
    """
    Given an argparse.ArgumentParser instance, return its argument flags in a space-separated
    string.
    """
    def parser_flags(parser):
    return''.join('--{0}'.format(f) for f in parser._flags)
```

Incomplete Generation

This label is applied when the LLM LLM generates no code but instead produces for example a "pass" statement or an empty function or it stops completing the code in the middle of the function. Listing A.10 generated by CodeGen is an example of this category.

Listing A.10 Code generated by CodeGen and labeled as "Incomplete Generation".

```
def write_configuration(config_filename, rendered_config, mode=0o600, overwrite=False):
        """
        Given a target config filename and rendered config YAML, write it out to file. Create any
containing directories as needed. But if the file already exists and overwrite is False,
abort before writing anything.
        """
        if os.path.exists(config_filename) and overwrite:
                print("Overwriting " +
                #
```

Non-Prompted Consideration (NPC)

This label is applied when the LLM adds statements to the code that are unrelated to the task specification, leading to errors or serving as the root cause of bugs. For instance, in Listing A.12 from the code generated by Codex, although the prompt does not specify sorting the flags, at line 15, the function sorts the list of flags before joining them, which is not in line with the given prompt.

Listing A.11 Code generated by CodeGen and labeled as “NPC”.

```
def parser_flags(parser):
    '''
    Given an argparse.ArgumentParser instance, return its argument flags in a space-separated
    string.
    '''
    def parser_flags(parser):
        flags = []
        for action in parser._actions:
                if isinstance(action, argparse._SubParsersAction):
                        for choice, subparser in action.choices.items():
                                flags.append(choice)
                                flags += parser_flags(subparser)
                else:
                        flags.append(action.option_strings[0])
        return " ".join(sorted(flags))
```

APPENDIX B AUGMENTED PROMPT ON LLAMA-2-CHAT

```
############
//prompt
############
{"role": "system", "content": "You are a Python
coding assistant. Always answer with Python code."},
{"role": "user", "content":
#Generate test cases for the following code:
‘‘‘python
def choose_num(x, y):
    if x > y:
        return −1
    if y % 2 == 0:
        return y
    if x == y:
        return −1
    return y − 1
‘‘‘
},

{"role": "assistant", "content":
‘‘‘python
#test cases
def test():
     assert choose_num(6,10) == 10
     assert choose_num(11,6) == −1
     assert choose_num(8,12) == 12
‘‘‘
},

{"role": "user", "content":
#The test function, test(), cannot detect the fault in the following code:
‘‘‘python
def choose_num(x, y):

    if x > y:
```

```
        return −1
    if y % 2 == 0:
        return y
    if x != y:
        return −1
    return y − 1
‘‘‘
#Provide a new test case to detect the fault in the prior code:
‘‘‘python
#test case
def test():
    assert
}

##################################
//Generated output of llama−2−chat
##################################
"Here is a new test case to test the fault in the previous code:"
choose_num(5, 5) == −1

‘‘‘python
def test():
     assertchoose_num(5, 5) == −1
‘‘‘
"This test case checks for the scenario where both x and y are equal. In the
modified code, the condition if x != y will evaluate to False, and the expected
result is −1. However, in the original code, the condition if x == y will evaluate
to True, and the incorrect result would be y − 1, which is 4 instead of −1."
```