Università della Svizzera italiana

Software Institute

Exploring the usage of pre-trained models for code-related tasks

Antonio Mastropaolo

Research Advisor

Prof. Dr. Gabriele Bavota

SEART

Dissertation Committee

Prof. Antonio Carzaniga	Università della Svizzera italiana, Switzerland
Prof. Michele Lanza	Università della Svizzera italiana, Switzerland
Prof. Davide Di Ruscio	University of L'Aquila, Italy
Dr. Alexey Svyatkovskiy	Google DeepMind, USA

Dissertation accepted on 15 May 2024

Research Advisor
Prof. Gabriele Bavota

Ph.D. Program Co-Director
Prof. Walter Binder

Ph.D. Program Co-Director
Prof. Stefan Wolf

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Antonio Mastropaolo
Lugano, 15 May 2024

Abstract

Developers are often faced with the challenge of writing high-quality code while meeting strong time constraints. Recent literature exploits Deep Learning (DL) models to support developers in code-related tasks. For example, DL-based approaches have been proposed to automate bug-fixing activities, code summarization, and code review. Some of these tasks require to work with both code and technical natural language (*e.g.,* code summarization), posing additional challenges in the training of DL models which must deal with bi-modal data. Our goal is to widen the support given to developers when dealing with code-related tasks characterized by technical natural language and code. To this extent, we started investigating the benefits brought by the "pretrain-then-finetune" paradigm when using DL models to automate code-related activities. The basic idea of this paradigm is to first pre-train the model with self-supervised tasks with the only goal of learning the languages of interest (*e.g.,* technical English and code). Then, the fine-tuning phase takes care of specializing the model for the specific task of interest (*e.g.,* code summarization). Given the positive results we achieved, we focused our research on two code-related tasks characterized by both code and natural language. The first is the already mentioned code summarization which consists in generating a code summary for a given piece of code at hand (*e.g.,* method, code snippet). In this context, we also present a novel metric aimed at assessing automatically generated code summaries. The second is the generation and injection of complete log statements in which the DL model takes as input a code component and is in charge of recommending to developers which log statements may be beneficial to inject, thus taking care of generating the log statement (including a meaningful log message) and inject it in the correct code location. Finally, given the increasing popularity of the GitHub Copilot code recommender, we ran an empirical study on a third task characterized by both code and natural language, namely code generation (*i.e.,* generate the code needed to implement a functionality described in natural language). In particular, we investigated Copilot's robustness in handling different yet semantically-equivalent natural language descriptions of the code to implement (prompts), showing its sensitivity to the wording used in the prompt (*i.e.,* minor changes to the prompt result in different code synthesized by Copilot).

Acknowledgments

Well, it seems I am about to conclude another chapter of my life and what a CHAPTER!. Let me be quite frank, my Ph.D. started in October 2020, right between the first and oncoming second waves of COVID-19. Beginning a new life abroad only to be confronted with such "uncontrollable" events—those beyond the reach of any willpower and perhaps left to fate—was hardly ideal. However, it wasn't all doom and gloom; I made it through, albeit picking up smoking as my bad habit along the way. But as they say, such is life. The journey was challenging, yet realizing the fantastic people around me made all the difference so much so that at a certain point persevere was the only thing to do.

Matteo and Rosalia, for instance, together we embarked on the DEVINTA project. Despite our different life phases, we clicked since the very beginning. We've become more than colleagues; we're friends who've spent countless hours working, joking, and, most memorably, traveling to Australia together. Speaking of which, our return flight saw us upgraded to business class, for free!

My journey also introduced me to Luca and Emad, both post-docs at the time, amazing colleagues. Luca and I, to this day, share countless moments. From gym sessions to dinner invitations from him and his fiancée, Claudia—"Anto, pizza tonight?"—Never skipped!. Luca and Claudia became such good friends that I even spent a couple of days in Luca's hometown of Benevento to critique his pizza-making skills in a wood-fired oven.

The SoftwarE Analytic Research Team (SEART) truly stands out, thanks to the visionary leadership of Gabriele. He's exceptional and always pushed me to aim for more and do better. This means a great deal to me! Gabriele's ability to foster growth and development in others is remarkable. I can confidently say Gabriele is in a league of his own, he's built different. Forget about those horror stories on student-advisor dynamics; Gabriele transforms that narrative. He's not just an advisor but also a colleague, a friend, and a mentor. He thought me several lessons for which I will always be grateful, but the most important one I've learned from Gabriele is what kind of academic I aspire to be!. Reflecting on mentorship, I've been incredibly fortunate to receive an exceptional support from numerous mentors throughout my academic journey—a support I hope will continue. During my time as a bachelor's and later a master's student, my first academic advisor, Rocco Oliveto, introduced me to the world of research. Thanks to him and Gennaro Parlato, a small but persistent spark of interest was kindled within me. This passion has grown steadily over the years, culminating in September 2023 when I decided to pursue Assistant Professor positions primarily in North America. Since making this decision, I have received unwavering encouragement from Gabriele, Max, Michele, Andy, Denys, Gennaro, and many others.

Their support was phenomenal. Gabriele and Max, even before the interviews started, told me, "Anto, get ready for a journey across continents; you'll be invited for interviews all over the places." Initially skeptical, I thought, "We'll see...," but indeed, I traveled extensively–

Singapore, Montreal, San Francisco, Williamsburg, not to mention the interviews I declined because my decision was already made. Thanks to their support when I asked reference letters, providing continuous feedback, and guidance, especially when choosing between two places I wanted to go, I am proud to announce that in August 2024, I'll begin my role as an Assistant Professor at William & Mary, Virginia (USA).

Sharing my journey is crucial because the role of those who guides and supports us often goes unrecognized. Therefore, I must express my profound gratitude to Gabriele, Max, Michele, Andy, Denys, Rocco and Gennaro. They saw something in me, even when I doubted myself, and their genuine intentions, have been invaluable. To these people, I owe!. Recognizing someone's potential before they see it in themselves is a rare gift. So, a heartfelt thank you to all of you, truly, for everything!

Earlier, I mentioned how exceptional our team is, a sentiment largely attributable to its current members. Ozren, Alessandro, and Alberto are fantastic people who contribute significantly both within and beyond our team dynamics. Alessandro, affectionately known as the small man, is not just a colleague but a dear friend and my scapegoat during stressful times. Despite the occasional fun I make of him, he's well aware of my respect and concern for him. I often find myself encouraging him, confident in the success of his Ph.D. because of his remarkable determination and passion. Ozren, or Oz as I like to call him, was once my gym "buddy" and trusted spotter for those heavy lift. Even though he's traded weightlifting for rock climbing, he remains my favorite buddy for adventures and fun. A special shoutout to Alberto, the latest addition to the SEART group, who provided invaluable insights when I was prepping for my interviews.

Furthermore, I must extend my gratitude to all the colleagues and friends I've had the pleasure of meeting on this incredible journey, who have been part of some truly memorable moments. A heartfelt thank you to Emanuela, Federica, Vittorina, Ciccio, Federico, Alejo, Bin, Camilo, and Carlos. I'd like to use these final words to express my deepest gratitude to my family, who are the cornerstone of my life. Firstly, I want to extend my heartfelt thanks to my mother, my father, my sister, my grandmothers, and my grandfather. The values and principles they have instilled in me have made me aware of the blessings in my life. Their unwavering presence and support, no matter what challenges I faced, truly embody the kind of family I wish for everyone. My mother, Antonella, is always the first person who comes to mind when I think of family. Words fail to capture the immense gratitude I have for all that she has done and continues to do for me. I cannot express enough admiration for her; she is my entry into this world, my best friend, and my invaluable treasure. Her support and love have been indispensable to me. She is even tattooed on my forearm! Thank you, Mom, for giving me your all, regardless of the situation. Now it's my turn to take care of you; you can take a step back and enjoy the fruits of your hard work. I am also aware that I could not have grown into the man I am today without the lessons my Dad taught me, lessons that he continues to impart. You Dad, together with Mom, have given me so much over the past 28 years that simply mentioning over the few lines I'm writing does not make enough justice to your love and support. This thesis represents the outcome of our effort!

Grazie Mamma e Grazie Papà, per avermi guidato fin qui. Godetevi il traguardo, perché questo, è il nostro targuardo!

Contents

Part I

Prologue

1

Introduction

The ever-increasing complexity of software systems often pushes developers to look for a helping hand either from a team-mate or, when not possible, by relying on (semi-)automated tools. In response to these needs, researchers have proposed recommender systems for software developers. Robillard *et al.* [RWZ10] defined such systems as "*software tools that can assist developers with a wide range of activities, from reusing code to writing effective bug reports*". The first generation of recommender systems for software developers was characterized by approaches built on top of limited and manually-crafted heuristics. For instance, Moreno *et al.* [MAS+13b] presented *JSummarizer*, an eclipse plug-in to automatically generate a natural language description of a Java class by using a set of pre-defined templates. Although empirical studies have shown the potential usefulness of these tools [MMTM12, MBDP+16, LYTH13], they suffer of generalizability issues. For example, the number of templates that can be manually defined to document Java classes is clearly limited, and unlikely to cover all possible coding scenarios.

Lately, the surge of software development data hosted on platforms such as GitHub, paved the way to a new class of approaches based on *data-driven* recommenders that can learn how to automate code-related tasks by "looking" at activities performed by real developers in open source projects. To give the reader a better idea of the quantity of software data that can be found on these code repositories, as of January 2023, GitHub counts more than 100 Million developers and more than 400 Million repositories[1]. Such a sheer amount of data unlocked the usage of deep learning (DL) models to support code-related tasks, such as automatic bug-fixing [TWB+19a, CKT+19, MRJ+19, HSN18], code summarization [LHWM20, HLWM20, LOZ+21] and code completion [SDFS20, SLH+21, CCP+21, SZFS19, LLZJ20, LWLK17], among others. To better understand how such techniques work, let us discuss the DL model proposed by Tufano *et al.* [TWB+19a] to automatically fix bugs in Java methods. The model learns bug-fixing patterns by being trained on ∼58k pairs of buggy and fixed methods mined from software repositories. In particular, these are methods which have been subject to bug-fixing activities and, as such, it is possible to retrieve from software repositories their version before (buggy) and after (fixed) the bug-fix. Other techniques fo-

[1] https://en.wikipedia.org/wiki/GitHub

cus on tasks characterized by bi-modal data in the form of technical natural language (*e.g.,* code comments) and source code. This is the case of code summarization (*i.e.,* generating a natural language summary of a given code) and code generation (*i.e.,* generating the code needed to implement a functionality described in natural language). Automating these tasks implies the capability of the model to deal at the same time with both natural and programming languages.

The emergence of pre-trained models has brought considerable progress, opening up a breadth of opportunities for the automation of code-related tasks. With "pre-trained models", we refer to DL models which are pre-trained with self-supervised tasks to learn characteristics of the language of interest (*e.g.,* English and Java). For example, a classic pre-training objective is the *masked language model*, consisting in randomly masking a certain percentage (*e.g.,* 15%) of the tokens composing the input sequence (*e.g.,* an English sentence, a Java method), asking the model to predict those tokens that have been masked. This equips pre-trained models with a re-usable body of knowledge about the languages of interest that can be exploited when the model is then fine-tuned (*i.e.,* specialized) to support a specific task. For example, in the case of code summarization, the fine-tuning will provide the model with examples of code components (input) and their natural language summary (expected output).

The idea of using the "pretrain-then-finetune" paradigm to support software engineering (SE) tasks has been firstly proposed by Robbes *et al.* [RJ19], which suggested it as a way to overcome the limited size of training datasets available for specific tasks (*e.g.,* sentiment analysis on software-related corpora such as Stack Overflow discussions). Since then, also due to the advent of the Transformer DL architecture [VSP+17], pre-trained models have been widely adopted in SE, with tools such as Microsoft's GitHub Copilot [cop] exemplifying the start of a new era in coding automation.

Despite the advancements made in automating code-related tasks through DL-based techniques, we observe that (i) there are still tasks, especially those characterized by bi-modal data (*e.g.,* code summarization, code generation), for which the support provided to developers is limited and would benefit from further research [LSZ+22]; (ii) the empirical evaluation of these generative DL models is extremely challenging since, in most cases, it is difficult to objectively assess the correctness/quality of the output they generate (*e.g.,* is a generated code summary a good description of the code provided as input?); and (iii) the advent of tools such as GitHub Copilot poses new questions about how software will be developed and whether new skills will be required to software developers (*e.g.,* the ability to provide proper prompts to the tool to maximize its effectiveness).

1.1 Thesis Statement

Given the aforementioned premises, we formulate our thesis as follow:

> *Pre-trained deep learning models can effectively support the automation of tasks characterized by both code and natural language. Novel metrics are needed to properly assess their performance in the context of generative tasks.*

To validate our thesis, we first investigate the effectiveness of pre-trained models for several code related tasks, including those characterized by bi-modal data. Then, we propose solutions aimed at boosting two specific code-related tasks: snippet-level code summarization (*i.e.,* the task of describing in natural language a snippet of code composed by an arbitrary number of statements) and log injection (*i.e.,* the task of synthesizing and injecting log statements in source code). We assess the proposed solutions both quantitatively and qualitatively, pointing to their strengths and limitations.

Then, based on our experience with the evaluation of code summarization techniques, we present a novel metric aimed at assessing whether a natural language summary is appropriate for a given code. We show that such a metric captures orthogonal aspects of code summary quality as compared to those employed in the state-of-the-art, and it is the one having the highest correlation with human judgment of summary quality.

Finally, we take GitHub Copilot as representative of a state-of-the-art pre-trained model for code generation, and study its sensitivity to the prompt provided as input (*i.e.,* the natural language text describing the code to generate), showing its lack of robustness when prompted with semantically equivalent (but different) code descriptions.

1.2 Research Contributions

The research contributions of this thesis can be grouped into three high-level categories: (i) the definition and experimentation of DL-based techniques exploiting pre-trained models to automate code-related tasks, especially those involving bi-modal data such as code summarization and log statement generation, detailed in Chapters 3, 6, 8 and 9; (ii) the introduction of a novel metric aimed at assessing the effectiveness of code summarization methods, discussed in Chapter 10; and (iii) an empirical assessment of the *robustness* of GitHub Copilot [cop] for the task of code generation, covered in Chapter 4. The research presented in this thesis has been conducted in the context of the DEVINTA ERC project [Dev].

1.2.1 Automating Code-related Tasks via Pre-trained Models

We studied the extent to which pre-trained DL models can be instantiated to automate code-related tasks, including bug-fixing, source code mutation, generation of assert statements, code summarization and log generation. We started by running a first empirical study aimed at exploring the capabilities of a pre-trained DL model for the automation of several code-related tasks by comparing it against state-of-the-art techniques. We demonstrated the advantages of the pre-training process for each task undertaken, concluding that it serves as a useful mechanism for knowledge transfer, ultimately leading to superior performance compared to prior DL methodologies. This work resulted in the following publications:

> **Studying the Usage of Text-To-Text Transfer Transformer to Support Code-Related Tasks**
> A. Mastropaolo, S. Scalabrino, N. Cooper, D.N. Palacio, D. Poshyvanyk, R. Oliveto, G.Bavota.. In *43rd International Conference on Software Engineering (**ICSE 2021**)*, 336–347

> **Using Transfer Learning for Code-Related Tasks**
> A. Mastropaolo, N. Cooper, D.N. Palacio, S. Scalabrino, D. Poshyvanyk, R. Oliveto, G.Bavota. In *IEEE Transactions on Software Engineering (**TSE 2022**)*, 4818-4837

Then, given the promising results we achieved, we devised techniques exploiting pre-trained models to (partially) automate code summarization. A first approach, aimed at summarizing code snippets, while a second one focused on comment completion (*i.e.,* automatically completing a comment partially written by a developer). This part of the thesis has been presented in the following publications:

> **An Empirical Study on Code Comment Completion**
> A. Mastropaolo, E. Aghajani, L. Pascarella, G. Bavota. In *37th International Conference on Software Maintenance and Evolution (**ICSME 2021**)*, 159-170

> **Towards Summarizing Code Snippets Using Pre-Trained Transformers**
> A. Mastropaolo, E. M. Ciniselli, R. Tufano, L. Pascarella, E. Aghajani, G. Bavota. In *32nd International Conference on Program Comprehension (**ICPC 2024**)*, To Appear, 12 pages.

Finally, we presented and empirically evaluated the first approach in the literature exploiting a pre-trained model to provide complete automation of log statements injection in Java programs:

> **Using Deep Learning to Generate Complete Log Statements**
> A. Mastropaolo, L. Pascarella, G. Bavota. In *44th International Conference on Software Engineering (**ICSE 2022**)*, 2279-2290

> **Log Statements Generation via Deep Learning: Widening the Support Provided to Developers**
> A. Mastropaolo, V. Ferrari, L. Pascarella, G.Bavota. In *Elsevier Journal of Systems and Software (**JSS 2023**)*,

1.2.2 Improving the Evaluation of Code Summarization Techniques

We proposed a new metric aimed at overcoming the limitations of state-of-the-art metrics usually adopted to assess code summarization techniques (*e.g.,* BLEU [PRWZ02], METEOR [BL05]). Ideally, software developers should be involved in assessing the quality of the generated summaries. However, in most cases, researchers rely on automatic evaluation metrics. These metrics are all based on the same assumption: *The higher the textual similarity between the generated summary and a reference summary written by developers, the higher its quality*. However, there are two reasons for which this assumption falls short: (i) reference summaries, *e.g.,* code comments collected by mining software repositories, may be of low quality or even outdated; (ii) generated summaries, while using a different wording than a reference one, could be semantically equivalent to it, thus still being suitable to document the code snippet.

We performed a thorough empirical investigation on the complementarity of different types of metrics in capturing the quality of a generated summary. Then, we proposed to address the limitations of existing metrics by considering a new dimension, capturing the extent to which the generated summary aligns with the semantics of the documented code snippet, independently from the reference summary. To this end, we presented a new metric based

on contrastive learning to capture said aspect. We empirically showed that the inclusion of this novel dimension enables a more effective representation of developers' evaluations regarding the quality of automatically generated summaries. This part of the thesis resulted in the following publication:

> **Evaluating Code Summarization Techniques: A New Metric and an Empirical Characterization**
> A. Mastropaolo, M. Ciniselli, G. Bavota, M. Di Penta. In *46th International Conference on Software Engineering (**ICSE 2024**)*, To Appear, 12 pages.

1.2.3 Evaluating the Robustness of DL-based techniques for Generating Code

We studied the *robustness* of DL-based code generators by focusing on GitHub Copilot, currently being the most popular code generator among practitioners. Indeed, while the usefulness of Copilot is evident, it is still unclear the extent to which semantic-preserving changes in the natural language description provided to the model have an effect on the generated code. In particular, we studied whether *different but semantically equivalent natural language descriptions result in the same recommended function*. A negative answer would pose questions on the robustness of DL-based code generators since it would imply that developers using different wordings to describe the same code would obtain different recommendations. Our results showed that modifying the description results in different code recommendations in ∼46% of cases. Also, differences in the semantically equivalent descriptions might impact the correctness of the generated code (±28%). This part of the thesis resulted in the following publication:

> **On the Robustness of Code Generation Techniques: An Empirical Study on GitHub Copilot**
> A. Mastropaolo, L. Pascarella, G. Guglielmi, M.Ciniselli, S.Scalabrino, R.Oliveto, G.Bavota. In *45th International Conference on Software Engineering (**ICSE 2023**)*, 2149-2160

1.3 Outline

This dissertation is organized into parts, which are further split into chapters as described below.

Part II revolves around empirical investigations about the usage of DL-based techniques for code-related tasks and software engineering automation. Chapter 2 discusses the literature related to the code-related tasks under investigation in this part of the thesis, namely bug fixing, source code mutation, generation of assert statements, and method-level source code summarization. Chapter 2 also discusses empirical studies focusing on the advantages and limitations of AI-driven solutions for software development. Chapter 3 presents our study investigating the viability of employing the "pretrain-then-finetune" approach to foster the automation of the tasks of interests. We compared our technique against the state-of-the-art techniques proposed at that time for solving the same tasks. Chapter 4 details instead our study on the robustness of GitHub Copilot.

Part III comprises two different chapters. Chapter 5 provides general concepts about logging and discusses the most recent advancement and techniques to automate logging activities. Chapter 6 introduces LEONID, the end-to-end solution we devised to equip developers with a method that enables complete automation of log statements injection.

Part IV is dedicated to the topic of code summarization, outlining the methodologies we have established to enhance this task, along with introducing SIDE, a novel metric designed to improve the evaluation of code summarization techniques. Chapter 7 detail the status of current literature in the field concerning snippet-level code summarization and methods to evaluate approaches for documenting the code. Chapter 8 presents our approach for snippet-level summarization, while Chapter 9 focuses on the comment completion task. Finally, Chapter 10 outlines our efforts to develop a more effective evaluation framework for assessing the quality of automatically generated code summaries.

Part V concludes the thesis and outlines directions for future work.

Appendices A, B, C, and D detail additional research conducted during the PhD that falls outside the scope of this thesis.

Part II

Empirical Investigations About the Usage of DL-based Solutions for Code-Related Tasks

In this part, we present two studies investigating, from different perspectives, the application of DL-based solutions to code-related tasks. Specifically, our first study (Chapter 3) presents an empirical analysis on the usage of transfer learning for code-related tasks. Particularly, we explore the capabilities of a Text-to-Text Transfer Transformer (T5) model [RSR+20] that has been trained to support four different code-related tasks, namely: *bug-fixing*, *injection of code mutants*, *generation of assert statements in test methods*, and *code summarization*. To this extent, we start by pre-training the T5 model using a large dataset consisting of 499,618 English sentences and 1,569,889 source code components (*i.e.,* Java methods). Subsequently, we refine the model's capabilities through two distinct approaches: fine-tuning in a single-task and in a multi-task framework. In the single-task approach, we develop four different models, each tailored for only one of the tasks under investigation. Conversely, the multi-task approach involves training a single model designed to handle various tasks simultaneously (thus investigating the boost in performance provided by transfer learning, if any).

Our findings reveal that the pre-training phase of the T5 model significantly enhances its effectiveness across all tasks, showcasing the value of the pre-training step in boosting overall performance. However, when it comes to fine-tuning for multiple tasks simultaneously, the benefits are not uniformly observed. Still, the T5 model outperforms the baseline models (state-of-the-art techniques when we run our study back in 2021) in all four tasks. The results of this work have been published in the two following papers:

> **Studying the Usage of Text-To-Text Transfer Transformer to Support Code-Related Tasks**
>
> Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David N. Palacio, Denys Poshyvanyk, Rocco Oliveto, Gabriele Bavota. In *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering (ICSE 2021)*, pp. 336-347

> **Using Transfer Learning for Code-Related Tasks**
>
> Antonio Mastropaolo, Nathan Cooper, David N. Palacio, Simone Scalabrino, Denys Poshyvanyk, Rocco Oliveto, Gabriele Bavota. In *Transaction on Software Engineering (TSE 2022)*, Volume 49(4), pp. 1580-1598

The second study (Chapter 4) has been instead conducted after the advent of GitHub Copilot [CTJ+21], which has changed the landscape of coding automation by providing unprecedented performance in complex tasks such as code generation, *i.e.,* providing the tool with a natural language description of the code to implement (prompt) expecting as output the required code. One open question related to the adoption of this tool is what is the impact of the wording used when defining the prompt on the generated source code. In other words, what happens if two developers use a slightly different (but semantically equivalent) wording to describe the code they want to be automatically implemented? Will they obtain the same method as output? Will the quality of the obtained code be affected? To answer this

question, we collected a set of 892 Java methods that are (i) accompanied by a Doc Comment for the Javadoc tool, and (ii) exercised by a test suite written by the project's contributors. Then, we considered the first sentence of the Doc Comments as a "natural language description" of the method and generate (both manually and automatically) paraphrases of this sentence providing both as input to Copilot. We found that in ∼46% of cases semantically equivalent but different method descriptions result in different code recommendations. We observed that some correct recommendations can only be obtained using one of the semantically equivalent descriptions as input. These results highlight the importance of providing a proper code description when asking DL-based recommenders to synthesize code. In the new era of AI-supported programming, prompt engineering is thus a fundamental skill for developers. This research has been presented in the following publication:

> **On the Robustness of Code Generation Techniques: An Empirical Study on GitHub Copilot**
>
> Antonio Mastropaolo, Luca Pascarella, Emanuela Guglielmi, Matteo Ciniselli, Simone Scalabrino, Rocco Oliveto, Gabriele Bavota. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE 2023)*, pp. 2149-2160

Before presenting the two studies, Chapter 2 discuss the literature related to this part of the thesis.

2

Background and Related Work

We start by discussing the literature related to the four tasks subject of the study presented in Chapter 3 (*i.e., bug-fixing*, *injection of code mutants*, *generation of assert statements in test methods*, and *code summarization*).

Then, we discuss empirical works investigating the strengths and weaknesses of AI-based solutions of software developers, since those are related to our investigation on the impact of prompting on the recommendations generated by GitHub Copilot (Chapter 4).

2.1 Automated Bug-Fixing

A variety of approaches have been developed for automatically fixing software bugs (see *e.g.,* [LNFW12, LGDVFW12, SDLLR15, PP09, GS10, CGM+13, NNN+13]). Given the focus of this thesis, we discuss only those that make use of DL-based methodologies.

While the underlying technique and DL architecture may vary, all these approaches rely on large training sets featuring buggy and fixed versions of code components which have been collected from bug-fixes commits performed by developers in open source repositories. The basic idea is to provide the model with the buggy version of a code (*e.g.,* a buggy Java method) asking it to generate its fixed version. In other words, the DL model is in charge of learning bug-fixing patterns from the given training set.

Former attempts to DL-based bug-fixing involved the usage of Neural Machine Translation (NMT) under the assumption that the bug-fixing problem can be addressed as a translation from buggy to fixed code [GPKS17, BKS18, LKB+18, HSN18, MRJ+19, CKT+19, TWB+19a, LPP+20b, JLT21]. The underlying DL architectures are in these cases sequence-to-sequence models mostly exploiting RNNs (Recurrent Neural Networks) [She18] or CNN (Convolutional Neural Networks) [ON15]. Gupta *et al.* [GPKS17] have been pioneers in this area with their DeepFix approach, an end-to-end technique leveraging DL for fixing prevalent programming errors in C. Their assessment of DeepFix, carried out on a dataset of 6,971 student-written erroneous C programs, underscored the effectiveness of the proposed methods, achieving complete fixes for 27% of the programs and partial fixes for an additional 19%. Mesbah *et al.* [MRJ+19] focused instead on build-time compilation failures

by presenting DeepDelta, an approach using NMT to fix the build. The input is represented by features characterizing the compilation failure (*e.g.,* type of error, AST path, etc.). As output, DeepDelta provides the AST changes needed to fix the error. In the presented empirical evaluation, DeepDelta correctly fixed 19,314 out of 38,788 (50%) compilation errors. Chen *et al.* [CKT+19] presented SequenceR, a sequence-to-sequence approach trained on over 35k single-line bug-fixes. SequenceR takes as input the buggy line together with its "abstract buggy context", meaning the relevant code lines from the buggy class. The output of the approach is the recommended fix for the buggy line. The approach, tested on a set of 4,711 bugs, was able to automatically fix 950 (∼20%) of them. Similar approaches have been proposed by Hata *et al.* [HSN18], Tufano *et al.* [TWB+19a] and Li *et al.* [LWJ+22].

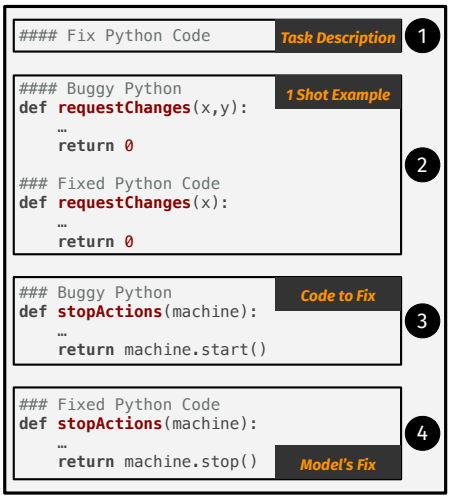
Other authors exploited GNN (Graph Neural Networks[ZCH+20] or tree-based representations [LWN20, LWN22a], with the idea that the tree-based structure inherent of the source code can be better represented exploiting GNNs [YL20, TLB+21, YL21, XWX22]. For instance, Tang *et al.* [TLB+21] presented Grasp, an end-to-end method for fixing bugs in Java programs that leverages a graph-to-sequence learning model. This technique transforms the source code into a graph format to preserve its structural integrity, thus providing the DL model with a more comprehensive understanding of the code's structure (as opposed to the flat sequence of tokens used in sequence-to-sequence models) to facilitate the synthesis of more accurate patches.

With the advent of the Transformer architecture [VSP+17] and its superior performance in tasks involving natural language [WSM+18], most of the proposals for novel bug-fixing tools shifted towards transformer-based solutions, possibly exploiting pre-trained models [MSC+21, JLT21, BHRV21, ZSX+21, YZH+22, NLN+22, LLF+22, JLL+23, ZSZ+23, YXF+23, ZPN+22a, WWJH23, ZLJX23, WXZ23]. In this context, we were among the first exploiting the advantages provided by a bimodal pre-training (*i.e.,* a pre-training step involving both natural language and code) on the automation of bug-fixing activities [MSC+21] — see Chapter 3. The T5 model we adopted, has then been also exploited in different fashions in several of the subsequent works (see *e.g.,* [YZH+22, ZPN+22a]). Following this, the availability of large Transformers already pre-trained on code such as CodeBERT [FGT+20] further pushed the research in this area [MH21a, KS20, HYS+22].

Finally, the latest generation of proposed techniques exploit Large Language Models (LLMs) such as CodeX and ChatGPT [cha]. These models undergo extensive pre-training across diverse datasets, encompassing billions of instances that include not just code files, but also a vast array of textual content sourced from the internet. Furthermore, the scale of these models significantly surpasses those previously discussed (*e.g.,* CodeBERT). This allows them to tackle complex tasks such as bug-fixing without the need for training them on large-scale datasets showcasing the task of interest. In the context of bug-fixing, this means avoiding the need to specialize the model on pairs representing a defective program and its corresponding fix. Instead, a proper prompt possibly accompanied by a few examples (*i.e.,* shots) of the target task may be sufficient. Fig. 2.1 presents a scenario showcasing 1-shot learning with OpenAI Codex [CTJ+21], with the model learning the task from only one example. In the figure, part ❶ of the prompt explains the model the task to perform (*i.e.,* bug-fixing in Python); part ❷ provides the 1-shot example; part ❸ displays the faulty

Python code that needs to be fixed, representing the code for which the practitioner would like to receive a fix. Finally, the model's proposed fix appears in ❹.

Figure 2.1. Example of bug fix using OpenAI Codex LLM [CTJ+21]



In this context, the ability to obtain good performance from the LLM mostly resides in crafting proper prompts (*i.e.,* prompt engineering) and several researchers studied how to exploit the potential of LLMs for bug-fixing [ZCG+22, JSG+23, FGM+23]. For instance, Zhang *et al.* [ZCG+22] examine the viability of employing LLMs, such as CodeX [CTJ+21], to repair bugs in python assignments. Specifically, the authors investigate the extent to which multi-modal prompts, iterative querying and few-shot selection help in producing correct patches for faulty python programs. Jin *et al.* [JST+23] presented InferFix, a comprehensive framework for fixing bugs in Java and C# programs that utilizes OpenAI Codex [CTJ+21] alongside a novel static analyzer.

Kong *et al.* [KCX+24] have made additional steps in enhancing conversation-driven bug-fixing with the introduction of ContrastRepair. This method capitalizes on the capabilities of the advanced chatbot, *i.e.,* ChatGPT, and utilizes contrastive testing pairs comprising both passing and failing test cases. The results demonstrate how ContrastRepair advances the state-of-the-art by effectively isolating the underlying cause of bugs through additional information supplemented in the form contrastive test pairs.

Also Xia *et al.* [XZ23] introduced a fully automated conversation-driven bug-fixing system harnessing ChatGPT [cha]. Their technique learns from both prior unsuccessful and viable patches, incorporating test failure data to offer real-time, adaptive feedback for generating new patches.

Contribution in the area. The introduction of DL in software engineering marked a significant transition in automated bug-fixing, moving from rule-based and meta-heuristic approaches [LNFW12] to techniques emulating human-like processes in generating patches. In this area, we contributed with one of the first applications of pre-trained transformers for generating more accurate patches [MSC+21, MCP+22] — see Chapter 3 — outperforming the leading technique of that time [TWB+19a].

2.2 Source Code Mutation

Another of the tasks we tackle in Chapter 3 is code mutation, namely the injection of artificial bugs in software programs, which are at the basis of mutation testing [JH10].

Several mutation frameworks are available in the literature, *e.g.,* μJava [MOK05], Jester [Jes00], Major [JJE14], Jumble [Two], PIT [PIT10], and javaLanche [SZ09]. These frameworks however are based on predefined catalogues of mutation operators. Brown *et al.* [BVLR17] were the first to propose a data-driven approach for generating code mutants, leveraging bug-fixes performed in software systems to extract syntactic-mutation patterns from the diffs of patches. Tufano *et al.* [TWB$^+$19b] built on this concept by presenting an approach using NMT to inject mutants that mirror real bugs. The idea is to reverse the learning process used for fixing bugs [TWB$^+$19a]: The model is trained to transform correct methods (*i.e.,* the method obtained after the bug-fixing activity) into buggy methods (before the bug-fix).

In another study, Tufano *et al.* [TKW$^+$20] present DeepMutation, an NMT model that incorporates a pre-trained component. This model underwent training with datasets comprised of bug-fix pairs sourced from numerous GitHub repositories, demonstrating its efficacy in creating significant mutants.

Tian *et al.* [TCZ$^+$22] suggested LEAM, a deep learning framework designed to generate mutants by learning from actual faults. The innovative aspect of this approach is its adaptation of the syntax-guided encoder-decoder architecture, aiming to maintain the syntactic accuracy of the produced mutants. The conducted analysis showcases LEAM's superior performance when compared to the earlier state-of-the-art tool, DeepMutation [TKW$^+$20].

Patra and Pradel introduced SemSeed [PP21], a novel technique that identifies mutation patterns from actual bug fixes and applies them to different code segments by evaluating the resemblance of identifiers and literals through the use of trained token embeddings.

Degiovanni and Papadakis [DP22, KDPT23] build upon the pre-trained language model of code CodeBERT [FGT$^+$20] and propose μBERT a framework for mutants generation. The achieved results highlighted that the fault revelation ability of μBERT is 17% higher than that of PiTest [PIT10], the state-of-the-art tool for mutation testing.

Garg *et al.* [GDM$^+$23b, GDM$^+$23a] tackled the problem of source code mutation testing by focusing on a specific type of mutation: those that can be identified through assertion inference. They proposed AIMS, an approach that utilizes a DL encoder-decoder model to learn how to represent mutants. Then, these representations are used to classify whether the mutation is likely to be detected by assertion inference.

Ibrahimzada *et al.* [ICRJ23] propose BugFarm a technique that leverages Foundation Models *i.e.,* GPT3.5-turbo, for generating mutants via bug injection. The authors demonstrated that BugFarm is more creative in generating mutants thanks to the power of LLMs in code synthesis in comparison to state-of-the-art techniques such as LEAM [TCZ$^+$22] and μBERT [KDPT23].

Contribution in the area. DL models are used to learn from large-scale training datasets mutants which are representative of real bugs. In this context, we were the first employing pre-trained transformers to mutate source code [MSC$^+$21] — Chapter 3, showing its superiority to the state-of-the-art technique at that time [TWB$^+$19b].

2.3 Generation of Assert Statements

The third code-related task subject of Chapter 3 is the generation of assert statements. This task entails the generation of assert statements conditioned to the particular context and demands of a given test. The general idea is to provide the model a test method that lacks an assert statement, alongside the focal method it is designed to evaluate (*i.e.,* the method under test), while being trained to predict how to generate an appropriate assert statement for the input test method.

We only discuss research addressing the generating assert statements, omitting DL-based techniques developed for synthesizing entire test cases [DRML22, TDS+20, ATA23].

Watson *et al.* [WTM+20] start from the work by Shamshiri *et al.* [Sha15], who observed that tools for the automatic generation of test cases such as Evosuite [FA11], Randoop [PDE07] and Agitar [agi] exhibit insufficiencies in the automatically generated assert statements. Thus, they propose ATLAS, an approach for generating syntactically and semantically correct unit test assert statements using NMT. Once provided with a test method missing an assert statement and the corresponding focal method it intends to test, ATLAS successfully generates assert statements that match those crafted by developers in ∼31% of cases.

In another investigation, Tufano *et al.* [TDSS22] explore the concept of transfer learning to automate the generation of assert statements. Specifically, they initially pre-train a sequence-to-sequence transformer model (*i.e.,* BART [LLG+20]) on a large-scale dataset that includes both source code and English language texts. Following this, the model undergoes fine-tuning specifically for the task of generating assert statements. The outcomes demonstrate the effectiveness of the proposed approach, with a success rate of 62% in producing assert statements comparable to those crafted by developers, thereby underscoring the advantages of applying transfer learning to coding tasks.

Zhang *et al.* [ZJW+23] also used pre-trained models for the generation of assert statements by presenting SAGA, a DL-based method exploiting CodeT5 [WWJH21] and a context-aware fine-tuning which incorporates contextual elements like developers' written summaries during its specialization. This approach has been demonstrated to result in improved performance when experimented on state-of-the-art benchmarks.

Contribution in the area. For this task, we showed how the "pretrain-then-finetune" training strategy helps in substantially boost performance when generating assert statements as compared to the state-of-the-art [WTM+20]. Subsequent works in the literature [TDSS22, ZJW+23] built on top of that idea.

2.4 Method-level Code Summarization

In the field of code summarization, we can distinguish between two different class of techniques: extractive [HAMM10, SPV11b, MAS+13a, RJAM17] and abstractive [SPV11a, MM16, JAM17, HLX+18a, HLWM20]. The former create a summary of a code component which includes information extracted from the component being summarized, while the latter may include in the generated summaries information that is not present in the code component to document. DL techniques have been used to support the generation of abstractive sum-

maries at different level of granularity such as method-level and snippet-level. The upcoming discussion focuses on the former.

The most straightforward approach to train a DL model for method-level summarization is to provide it with pairs ⟨method, comment⟩, where comment is a natural language summary for the method. This data is usually automatically collected by mining software repositories [HLX+18a, HLX+20a, LWZ+19, HLWM20, LHWM20, ACRC20, LZ18]. For example, Hu *et al.* [HLX+18a] employ a NMT model to automatically generate comments for a given Java method by collecting the training/testing data from ∼9k Java projects hosted on GitHub, and consider as "comment" the first sentence of the Javadoc linked to the method.

While the basic idea behind these approaches is the same, some of them adopt strategies aimed at boosting the model performance. One possibility is to enrich the model's input, considering additional information besides the method's source code [ZYY+19, LWZ+19, LJM19, HLX+20a, ZWZ+20a, XYSZ21, YXZ+20]. For instnace, LeClair *et al.* [LJM19] combine AST information with source code tokens, showing major improvements as compared to the best techniques available in the literature at that tim [HLX+18a, IKCZ16]. A similar idea has also been exploited by Hu *et al.* [HLX+20a]. Later on, Haque *et al.* [HLWM20] further enriched the input provided to the model by aggregating: (i) the source code of the method, as a flattened sequence of tokens representing the method; (ii) its AST representation; and (iii) the "file context", meaning the code of every other method in the same file. The authors show that adding the contextual information as one of the inputs substantially improves the BLEU score obtained by deep learning techniques.

Other authors experimented with different neural network architectures, such as tree-based DL methods [SKY+19, LHWM20, HRC19], Deep Reinforcement Learning (DRL) [WZY+18, WZZX20, WZS+20] and ensamble methods [LBM21]. For example, Wan *et al.* [WZY+18] introduce a methodology where sequential segments of code snippets, which represent methods, are inputted into a DRL actor-critic network that once properly trained using the BLEU metric [PRWZ02] as a reward function, provides high-quality code summaries, achieving new state-of-the-art results in the field.

As already discussed for tasks such as bug-fixing, the advent of the Transformer architecture with its attention mechanism [VSP+17] pushed researchers to propose Transformer-based solutions for code summarization [LZJ20, ACRC20, WZL+20, GRL+21] as well as techniques exploiting variations of the vanilla Transformer [LOZ+21, GGH+23, WZZX20]. In this context, Gao *et al.* [GGH+23] proposed SG-Trans, a method that refines the original self-attention mechanism [VSP+17] specializing it to more accurately represent the structural characteristics of code, covering local structures at both the token and statement levels. This adaptation allows for a more nuanced understanding and modeling of code structure, enhancing the model's overall effectiveness in processing and interpreting code for summarization.

Finally, recent works exploit LLMs with specific prompts to automatically document code at function-level [HZD+22, GHLH21, SS20, TMC+21, YLGS, AD22, APDB24].

For example, Yun *et al.* [YLGS] examine in-context learning methods for project-specific code summarization. In particular, they discovered that utilizing ChatGPT with contextual examples (*i.e.,* shots) —— specifically, code snippets from the same project —— significantly

enhances the ability of the chatbot in generating good, high-quality summaries, setting new benchmarks for the field. On a similar note, Ahmed and Devambu [AD22] delved into the effectiveness of Codex in performing code summarization through in-context, few-shot learning. They discovered that as few as ten project-specific shots, used to steer the model towards generating code summaries, surpass the performance of conventional models fine-tuned on thousands of examples when generating meaningful summaries for *Java* and Python methods.

Contribution in the area. In our exploration of the role pre-trained models play in facilitating code-related tasks (Chapter 3), we demonstrated that our bi-modal pre-trained technique significantly surpassed the then-current state of the art [HLWM20].

2.5 Strength and Weaknesses of AI-driven Solutions for Software Developers

Chapter 4 features a study on the robustness of GitHub Copilot for the code generation task (*i.e.,* generating code starting from a natural language description). We thus review other works in the literature studying Copilot from different perspectives.

Most of the previous research aimed at evaluating the impact of GitHub *Copilot* on developers' productivity and its effectiveness (in terms of correctness of the provided solutions). Imai [Ima22] investigated to what extent *Copilot* is actually a valid alternative to a human pair programmer. They observed that *Copilot* results in increased productivity (*i.e.,* number of added lines of code), but decreased quality in the produced code. Ziegler *et al.* [ZKL+22] conducted a case study in which they investigated whether usage measurements about *Copilot* can predict developers' productivity. They found that the acceptance rate of the suggested solutions is the best predictor for perceived productivity. Vaithilingam *et al.* [VZG22] ran an experiment with 24 developers to understand how *Copilot* can help developers complete programming tasks. Their results show that *Copilot* does not improve the task completion time and success rate. However, developers report that they prefer to use *Copilot* because it recommends code that can be used as a starting point and saves the effort of searching online.

Nguyen and Nadi [NN22] used LeetCode questions as input to *Copilot* to evaluate the solutions provided for several programming languages in terms of correctness — by running the test cases available in LeetCode — and understandability — by computing their Cyclomatic Complexity and Cognitive Complexity [Cam18]. They found notable differences among the programming languages in terms of correctness (between 57%, for Java, and 27%, for JavaScript). On the other hand, *Copilot* generates solutions with low complexity for all the programming languages.

Two previous studies aimed at evaluating the security of the solutions recommended by *Copilot*. Pearce *et al.* [PAT+21] investigated the likelihood of receiving from *Copilot* recommendations including code affected by security vulnerabilities. They observed that vulnerable code is recommended in 40% of cases out of the completion scenarios they experimented with. On a similar note, Sobania *et al.* [SBR21] evaluated GitHub *Copilot* on

standard program synthesis benchmark problems and compared the achieved results with those from the genetic programming literature. The authors found that the performance of the two approaches are comparable. However, approaches based on genetic programming are not mature enough to be deployed in practice, especially due to the time they require to synthesize solutions.

Other researchers focused on exploring different AI-driven solutions such as ChatGPT [cha] by examining aspects revolving around the correctness of the generated code [LXWZ24], code explanation [CHC+23], the safety of ChatGPT's generated code [EGOK+23], for which tasks developers use ChatGPT [TMP+24], and how the usage of these technologies impacts software engineering education [DB23].

Contribution in the area. The undeniable impact of AI in automating software engineering practices is evident, with industry leaders like Microsoft and OpenAI enhancing AI models to unprecedented levels. Yet, this advancement calls for a critical evaluation of these AI-driven recommenders for software engineering tasks. In this context, we explored the aspect of *Robustness*, particularly how different but semantically equivalent prompts impact the performance of *Copilot* in code generation (Chapter 4).

3

Towards Automating Code-Related Tasks via Pre-trained Models of Code

Recent years have seen the rise of *transfer learning* in the field of natural language processing. The basic idea is to first pre-train a model on a large and generic dataset by using a self-supervised task, *e.g.,* masking tokens in strings and asking the model to guess the masked tokens. Then, the trained model is fine-tuned on smaller and specialized datasets, each one aimed at supporting a specific task. In this context, Raffel *et al.* [RSR+20] proposed the T5 (Text-To-Text Transfer Transformer) model, pre-trained on a large natural language corpus and fine-tuned to achieve state-of-the-art performance on many tasks, all characterized by text-to-text transformations and bi-modal data manipulation.

Drawing inspiration from recent advancements in the NLP field, we decided to empirically investigate the potential of a T5 model when pre-trained and fine-tuned to support four code-related tasks characterized by text-to-text transformations. In particular, we started by pre-training a T5 model using a large dataset consisting of 499,618 English sentences and 1,569,889 source code components (*i.e.,* Java methods). Then, we fine-tuned the model using four datasets from previous work with the goal of supporting four tasks:

Automatic bug-fixing. We used the dataset by Tufano *et al.* [TWB+19a], composed of instances in which the "input string" is represented by a buggy Java method and the "output string" is the fixed version of the same method.

Injection of code mutants. This dataset is also by Tufano *et al.* [TWB+19b], and features instances in which the input-output strings are reversed as compared to automatic bug-fixing (*i.e.,* the input is a fixed method, while the output is its buggy version).

Generation of assert statements in test methods. We use the dataset by Watson *et al.* [WTM+20], composed of 158,096 instances in which the input string is a representation of a test method without an assert statement and a focal method it tests (*i.e.,* the main production method tested), while the output string encodes an appropriate assert statement for the input test method.

Code Summarization. We use the dataset by Haque *et al.* [HLWM20] which features 2.1M Java methods paired with summaries.

Our analysis also seeks to uncover the tangible benefits (if any) that transfer learning pro-

vides for code-related. Such observation holds for both (i) the pre-training phase, that should provide the model with general knowledge about a language of interest (*e.g.,* Java) being at the core of the tasks to automate (*e.g.,* bug-fixing); and (ii) the multi-task fine-tuning, that should allow the model to exploit knowledge acquired when trained for a specific task (*e.g.,* bug-fixing) also for the automation of other tasks (*e.g.,* generation of assert statements), thus possibly boosting the overall performance in all the tasks.

To this extent, we assess the performance of the T5 in the following scenarios:

- **No Pre-training:** We do not perform any pre-training step. We directly fine-tune four different T5 models, each one supporting one of the four tasks we experiment with.

- **Pre-training single task:** We first pre-train the T5 model on the dataset presented in Table 3.1. Then, starting from it, we fine-tune four models, one for each single task.

- **Pre-training Multi-Task**: Lastly, we fine-tune the pre-trained model using a multi-task learning framework in which we train a single model to support all four code-related tasks. We experiment with two different multi-task fine-tunings: (i) the percentage of training instances from each of the four tasks is proportional to the size of their training dataset; (ii) the percentage of training instances is the same for all four tasks (*i.e.,* 25% per task).

In total, this results in the training, hyperparameters tuning, and testing of ten different models. Note that the choice of the four tasks subject of our study (*i.e., bug-fixing*, *mutants injection*, *asserts generation*, and *code summarization*) is dictated by the will of experimenting with tasks that use, represent, and manipulate code in different ways. In particular, we include in our study tasks aimed at (i) transforming the input code with the goal of changing its behavior (*bug-fixing* and *mutants injection*); (ii) "comprehending code" to verify its behavior (*asserts generation*); and (iii) "comprehending code" to summarize it in natural language (*code summarization*). Also, following what has been done in the original datasets from previous work, the four tasks involve abstracted source code (*bug-fixing* [TWB$^+$19a], *mutants injection* [TWB$^+$19b], and *asserts generation* [WTM$^+$20]), raw source code (*asserts generation* [WTM$^+$20] and *code summarization* [HLWM20]), and natural language (code summarization [HLWM20]). Such a mix of tasks helps in increasing the generalizability of our findings.

We also aim at assessing the generalizability of our models by looking at the level of data snooping among our training and test datasets.

Our results confirm that the T5 can substantially boost the performance on all four code-related tasks. For example, when the T5 model is asked to generate assert statements on raw source code, ∼70% of test instances are successfully predicted by the model, against the 18% of the original baseline [WTM$^+$20]. Also, we show that the pre-training is beneficial for all tasks, while the multi-task fine-tuning does not consistently help in improving performance. Finally, our datasets analysis confirm the generalizability of the tested models. The code and data used in this work are publicly available [repg].

3.1 Text-to-Text-Transfer-Transformer

The T5 model has been introduced by Raffel *et al.* [RSR+20] to support multitask learning in Natural Language Processing (NLP). The idea is to reframe NLP tasks in a unified text-to-text format in which the input and output are always text strings. For example, a single model can be trained to translate across languages and to autocomplete sentences. This is possible since both tasks can be represented in a text-to-text format (*e.g.,* in the case of translation, the input is a sentence in a given language, while the output is the translated sentence). T5 is trained in two phases: *pre-training*, which allows defining a shared knowledge-base useful for a large class of sequence-to-sequence tasks (*e.g.,* guessing masked words in English sentences to learn about the language), and *fine-tuning*, which specializes the model on a specific downstream task (*e.g.,* learning the translation of sentences from English to German). We briefly overview the T5 model and explain how we pre-trained and fine-tuned it to support the four said code-related tasks. Finally, we describe the decoding strategy for generating the predictions.

3.1.1 An Overview of T5

T5 is based on the transformer model architecture that allows handling a variable-sized input using stacks of self-attention layers. When an input sequence is provided, it is mapped into a sequence of embeddings passed into the encoder. The T5, in particular, and a transformer model [VSP+17], in general, offer two main advantages over other state-of-the-art models: (i) it is more efficient than RNNs since it allows to compute the output layers in parallel, and (ii) it is able to detect hidden and long-ranged dependencies among tokens, without assuming that nearest tokens are more related than distant ones. This last property is particularly relevant in code-related tasks since a variable declaration may be distant from its usage. Five different versions of T5 have been proposed [RSR+20]: *small*, *base*, *large*, *3 Billion*, and *11 Billion*. These variants differ in terms of complexity, with the smaller model ($T5_{small}$) having 60M parameters against the 11B of the largest one ($T5_{11B}$). As acknowledged by the authors [RSR+20], even if the accuracy of the most complex variants is higher than the less complex models, the training complexity increases with the number of parameters. Considering the available computational resources, we decided to use the simplest $T5_{small}$ model.

$T5_{small}$ architectural details. The $T5_{small}$ architecture is characterized by six blocks for encoders and decoders. The feed-forward networks in each block consist of a dense layer with an output dimensionality (d_{ff}) of 2,048. The *key* and *value* matrices of all attention mechanisms have an inner dimensionality (d_{kv}) of 64, and all attention mechanisms have eight heads. All the other sub-layers and embeddings have a dimensionality (d_{model}) of 512.

3.1.2 Pre-training of T5

In the *pre-training* phase we use a self-supervised task similar to the one used by Raffel *et al.* [RSR+20], consisting of masking tokens in natural language sentences and asking the model to guess the masked tokens. However, we did not perform the pre-training by only using natural language sentences, since all the tasks we target involve source code.

We use a dataset composed of both (technical) natural language (*i.e.,* code comments) and source code. To obtain the dataset for the pre-training we start from the CodeSearchNet dataset [HWG+19] which provides 6M functions from open-source code. We only focus on the ∼1.5M methods written in Java, since the four tasks we aim at supporting are all related to Java code and work at method-level granularity (*e.g.,* fixing a bug in a method, generating the summary of a method, etc.).

Then, since for three of the four tasks we support (*i.e., automatic bug-fixing* [TWB+19a], *generation of assert statements* [WTM+20], and *injection of code mutants* [TWB+19b]) the authors of the original papers used an abstracted version of source code, we used the `src2abs` tool by Tufano [TWB+19a] to create an abstracted version of each mined Java method. In the abstraction process, special tokens are used to represent identifiers and literals of the input method. For example, the first method name found (usually the one in the method signature) will be assigned the `METHOD_1` token, likewise the second method name (*e.g.,* a method invocation) will be represented by `METHOD_2`. This process continues for all the method and variable names (`VAR_X`) as well as the literals (`STRING_X`, `INT_X`, `FLOAT_X`). Basically, the abstract method consists of language keywords (*e.g.,* `for`, `if`), separators (*e.g.,* "(", ";", "}") and special tokens representing identifiers and literals. Comments and annotations are removed during abstraction. Note that, since the tool was run on Java methods in isolation (*i.e.,* without providing it the whole code of the projects they belong to), `src2abs` raised a parsing error in ∼600k of the ∼1.5M methods (due *e.g.,* to missing references), leaving us with ∼900k abstracted methods. We still consider such a dataset as sufficient for the pre-training.

The CodeSearchNet dataset does also provide, for a subset of the considered Java source code methods, the first sentence in their Javadoc. We extracted such a documentation using the `docstring_tokens` field in CodeSearchNet, obtaining it for 499,618 of the considered methods. We added these sentences to the pre-training dataset. This whole process resulted in a total of 2,984,627 pre-training instances, including raw source code methods, abstracted methods, and code comment sentences. In the obtained dataset there could be duplicates between (i) different raw methods that become equal once abstracted, and (ii) comments re-used across different methods. Thus, we remove duplicates, obtaining the final set of 2,672,423 instances reported in Table 3.1. This is the dataset we use for pre-training the T5 model, using the BERT-style objective function Raffel *et al.* used in their experiments and consisting of randomly masking 15% of tokens (*i.e.,* words in comments and code tokens in the raw and abstracted code).

Table 3.1. Datasets used for the pre-training of T5.

Data sources	**Instances**
Source code	1,569,773
Abstracted source code	766,126
Technical natural language	336,524
Total	2,672,423

Finally, since we pre-train and fine-tune the models on a software-specific dataset, we cre-

ate a new *SentencePiece* model [KR18] (*i.e.,* a tokenizer for neural text processing) by training it on the entire pre-training dataset so that the T5 model can properly handle the *Java* language and its abstraction. This model implements subword units (*e.g.,* byte-pair-encoding BPE) and unigram language model [Kud18] to alleviate the open vocabulary problem in neural machine translation. The pre-training of the models has been performed for 250k steps which, using a batch size of 128 results in ∼32M of masked code instances processed that, given the size of the pre-training dataset (see Table 3.1) correspond to ∼12 epochs.

3.1.3 Fine-tuning of T5

We detail the process used to fine-tune the T5 model. Before explaining how the training instances are represented within each fine-tuning dataset, it is important to clarify that both in the pre-training and in the fine tuning the T5 can handle any sort of training instance as long as it can be formulated as a text-to-text transformation. Indeed, the T5 represents each training dataset as a $N \times 2$ matrix, where N is the number of instances in the dataset and the 2 dimensions allow to express the input text and the expected output text. In the case of pre-training, the input text is an instance (*i.e.,* a raw method, an abstract method, or a Javadoc comment) in which 15% of tokens have been masked, while the output text represents the correct predictions for the masked tokens. In the four downstream tasks, instead, the text-to-text pairs are represented as explained in the following.

3.1.4 Fine-tuning dataset

We describe the datasets we use for *fine-tuning* the model for the four targeted tasks. The datasets are summarized in Table 3.2. The number of training steps performed for the different tasks is proportional to the size of their training dataset. Indeed, we aim at ensuring that the same number of "epochs" is performed on each training dataset. Thus, smaller training datasets require a lower number of steps to reach the same number of epochs of larger datasets. In particular, we used 1.75M fine-tuning steps for the *multi-task* setting ∼90 epochs) and we scaled the others proportionally to reach the same number of epochs (*e.g.,* ∼1.41M for the *code summarization* task).

Table 3.2. Task-specific datasets used for fine-tuning T5.

Task	**Dataset**	**Training-set**	**Evaluation-set**	**Test-set**
Automatic Bug-Fixing	BF_{small} [TWB+19a]	46,680	5,835	5,835
	BF_{medium} [TWB+19a]	52,364	6,546	6,545
Injection of Code Mutants	MG_{ident} [TWB+19b]	92,476	11,560	11,559
Generation of Asserts in Test	AG_{abs} [WTM+20]	126,477	15,809	15,810
	AG_{raw} [WTM+20]	150,523	18,816	18,815
Code Summarization	*CS* [HLWM20]	1,953,940	104,272	90,908
Total		2,422,460	162,838	149,472

Automatic Bug Fixing (BF). We use the dataset by Tufano *et al.* [TWB+19a]. To build this dataset, the authors mined ∼787k bug-fixing commits from GitHub, from which they extracted ∼2.3M BFPs. After that, the code of the BFPs is abstracted to make it more suitable for the NMT model (*i.e.,* to reduce the vocabulary of terms used in the source code identifiers and literals). The abstraction process is depicted in Fig. 3.1.

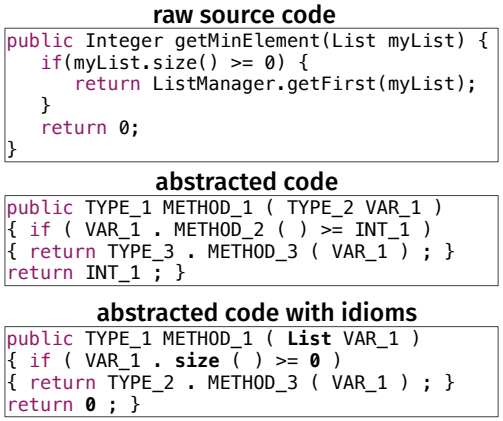


Figure 3.1. Abstraction process [TWB+19a]

The top part of the figure represents the raw source code to abstract. The authors use a Java lexer and a parser to represent each method as a stream of tokens, in which Java keywords and punctuation symbols are preserved and the role of each identifier (*e.g.,* whether it represents a variable, method, etc.) as well as the type of a literal is discerned.

IDs are assigned to identifiers and literals by considering their position in the method to abstract: The first variable name found will be assigned the ID of VAR_1, likewise the second variable name will receive the ID of VAR_2. This process continues for all identifiers as well as for the literals (*e.g.,* STRING_X, INT_X, FLOAT_X). The output of this stage is the code reported in the middle of Fig. 3.1 (*i.e.,* abstracted code). Since some identifiers and literals appear very often in the code (*e.g.,* variables `i`, `j`, literals `0`, `1`, method names such as `size`), those are treated as "idioms" and are not abstracted (see bottom part of Fig. 3.1, idioms are in bold). Tufano *et al.* consider as idioms the top 0.005% frequent words in their dataset. During the abstraction a mapping between the raw and the abstracted tokens is maintained, thus allowing to reconstruct the concrete code from the abstract code generated by the model.

The set of abstracted BFPs has been used to train and test the approach. The authors build two different sets, namely BFP_{small}, only including methods having a maximum length of 50 tokens (for a total of 58,350 instances), and BFP_{medium}, including methods up to 100 tokens (65,455).

We train the model to predict the fixed versions, m_f, given the buggy versions, m_b. Given the presence of two datasets, we divide the BF task in two sub-tasks, BF_{small} and BF_{medium}, depending on the size of the involved methods [TWB+19a].

Injection of Code Mutants (MG). For the MG task we exploited one of the two datasets

provided by Tufano *et al.* [TPW+19]: MG_{ident} and $MG_{ident-lit}$. In both datasets each instance is represented by a triple $\langle m_f, m_b, M \rangle$, where, similarly to the BF datasets, m_b and m_f are the buggy and fixed version of the snippet, respectively, and M represents the mapping between the abstracted tokens and the code tokens. The first dataset (MG_{ident}) represents the most general (and challenging) case, in which the mutated version, m_b, can also contain new tokens (*i.e.,* identifiers, types, or method names) not contained in the version provided as input (m_f). $MG_{ident-lit}$, instead, only contains samples in which the mutated version contains a subset of the tokens in the non-mutated code. In other words, $MG_{ident-lit}$ represents a simplified version of the task. For this reason, we decided to focus on the most general scenario and we only use the MG_{ident} dataset.

Generation of Assertions in Test Methods (AG). For the AG task we used the dataset provided by Watson *et al.* [WTM+20] containing triplets $\langle T, TM_n, A \rangle$, where T is a given test case, TM_n is the *focal* method tested by T, *i.e.,* the last method called in T before the assert [QBO+14], and A is the assertion that must be generated (output). For such a task, we use two versions of the dataset: AG_{raw}, which contains the raw source code for the input ($T + TM_n$) and the output (A), and AG_{abs}, which contains the abstracted version of input and output, *i.e.,* $src2abs(T + TM_n)$ and $src2abs(A)$, respectively. These are the same datasets used in the original paper.

Code Summarization (CS). For code summarization, we exploited the dataset provided by Haque *et al.* [HLWM20] containing 2,149,120 instances, in which each instance is represented by a tuple $\langle S, A_S, C_S, D \rangle$, where S represents the raw source code of the method, A_S is its AST representation, C_S is the code of other methods in the same file, and D is the summary of the method, *i.e.,* the textual description that the model should generate [HLWM20]. For this specific task, we consider a variation of the original dataset to make it more coherent with the performed pre-training. In particular, since in the pre-training we did not use any AST representation of code, we decided to experiment with the T5 model in a more challenging scenario in which only the raw source code to summarize (*i.e., S*) is available to the model. Therefore, the instances of our dataset are represented by tuples $\langle S, D \rangle$: We train our model to predict D given only S.

3.1.5 Decoding Strategy

Once the models have been trained, different decoding strategies can be used to generate the output token streams. T5 allows to use both *greedy decoding* and *Beam-search*. When generating an output sequence, the greedy decoding selects, at each time step t, the symbol having the highest probability. The main limitation of greedy decoding is that it only allows the model to generate one possible output sequence (*e.g.,* one possible bug fix) for a given input (*e.g.,* the buggy method).

Beam-search is an alternative decoding strategy previously used in many DL applications [Gra12, BLBV13, BCB14, RVY14]. Unlike greedy decoding, which keeps only a single hypothesis during decoding, beam-search of order K, with $K > 1$, allows the decoder to keep K hypotheses in parallel: At each time step t, beam-search picks the K hypotheses (*i.e.,* sequences of tokens up to t) with the highest probability, allowing the model to output K

possible output sequences.

We used Beam-search to provide several output sequences given a single input, and report results with different K values. It is worth noting that having a large K increases the probability that one of the output sequences is correct, but, on the other hand, it also increases the cost of manually analyzing the output for a user (*i.e.,* a developer, in our context).

3.1.6 Data Balancing for the multi-task model

The datasets we use for fine-tuning have different sizes, with the one for code summarization dominating the others (see Table 3.2). This could result in an unbalanced effectiveness of the model on the different tasks. In our case, the model could become very effective in summarizing code and less in the other three tasks. However, as pointed out by Arivazhagan *et al.* [ABF+19], there is no free lunch in choosing the balancing strategy when training a multi-task model, with each strategy having its pros and cons (*e.g.,* oversampling of less represented datasets negatively impacts the performance of the most representative task). For this reason, we decide to experiment with both strategies. In the first strategy, we follow the true data distribution when creating each batch. In other words, we sample instances from the tasks in such a way that each batch during the training has a proportional number of samples accordingly to the size of the training dataset. For the second strategy, we train a multi-task pre-trained model using a balanced sampling strategy. In other words, we feed the T5 model with batches of data having exactly the same number of samples per task randomly selected during the fine-tuning.

The results we obtained confirm the findings of Arivazhagan *et al.* [ABF+19]. In particular, when using the first training sampling strategy (*i.e.,* proportional sampling), the performance of the tasks having a large training dataset (*i.e.,* AG_{abs}, AG_{raw}, CS) had a boost. In contrast, when using the second strategy (*i.e.,* balanced sampling), the performance increases for those tasks whose training dataset is small with, however, a price to pay for the other three tasks. Nonetheless, since the observed differences in performance are not major and each strategy has its pros and cons, we decided to discuss in this thesis the results achieved using the proportional sampling schema, as we did in [MSC+21].

The results of the proportional sampling are available in our replication package [repg].

Table 3.3. Baselines and evaluation metrics for the tasks.

Task	Baseline	Accuracy@K	BLEU-n	ROUGE LCS
Automatic Bug-Fixing	[TWB+19a]	$\{1, 5, 10, 25, 50\}$	-	-
Injection of Code Mutants	[TWB+19b]	$\{1\}$	$\{A\}$	-
Generation of Asserts in Test	[WTM+20]	$\{1, 5, 10, 25, 50\}$	-	-
Code Summarization	[HLWM20]	-	$\{1, 2, 3, 4, A\}$	$\{P, R, F\}$

3.2 Research Questions and Context

We aim at investigating the performance of the T5 model on four code-related tasks: *Automatic bug-fixing, Injection of code mutants, Generation of Asserts in Tests and Code Summarization*. The focus of our evaluation is on (i) investigating the extent to which *transfer learning* is beneficial when dealing with code-related tasks, studying the impact on performance of both pre-training and multi-task learning; and (ii) comparing the obtained results with representative state-of-the-art techniques. The *context* is represented by the datasets introduced in Section 3.1.3, *i.e.,* the ones by Tufano *et al.* for bug fixing [TWB+19a] and injection of mutants [TWB+19b], by Watson *et al.* for assert statement generation [WTM+20], and by Haque *et al.* for code summarization [HLWM20]. We aim at answering the following research questions (RQs):

- **RQ_1:***What are the performances of the T5 model when supporting code-related tasks?* With RQ_1 we aim at understanding the extent to which T5 can be used to automate code-related tasks, investigating the performance achieved by the model on the four experimented tasks. In the context of RQ_1, we also investigate the impact of transfer learning on performance:

  - **$RQ_{1.1}$:** *What is the role of pre-training on the performances of the T5 model for the experimented code-related tasks?*

    With $RQ_{1.1}$ we aim at investigating the boost in performance (if any) brought by pre-training the models on a software-specific dataset.

  - **$RQ_{1.2}$:** *What is the role of multi-task learning on the performances of the T5 model for the experimented code-related tasks?* $RQ_{1.2}$ analyzes the influence of the *multi-task learning* (*i.e.,* training a single model for all four tasks) on the model's performance.

- **RQ_2:***What are the performances of T5 as compared with state-of-the-art baselines?* In RQ_2 we compare the performances achieved by the T5 model against the ones achieved by the baseline approaches. In this regard, we run T5 on the same test sets used in the four original papers presenting automated solutions for the code-related tasks we target.

3.2.1 Data Collection and Analysis

As explained in Section 3.1.3, we experimented with different variants of the T5: (i) *no pre-training* (*i.e.,* four models each fine-tuned for one of the supported tasks, without any pre-training); (ii) *pre-training single task* (*i.e.,* four models each fine-tuned for one of the supported tasks, with pre-training); and (iii) *pre-training multi-task* (*i.e.,* one model pre-trained and fine-tuned for all four tasks). These nine models have all been run on the test sets made available in the works presenting our four baselines and summarized in Table 3.2. Once obtained the predictions of the T5 models on the test sets related to the four tasks, we compute

the evaluation metrics reported in Table 3.3. We use different metrics for the different tasks, depending on the metrics reported in the papers that introduced our baselines.

Accuracy@K measures the percentage of cases (*i.e.,* instances in the test set) in which the sequence predicted by the model equals the oracle sequence (*i.e.,* perfect prediction). Since we use beam-search, we report the results for different K values (*i.e.,* 1, 5, 10, 25, and 50), as done in [TWB+19a] (BF) and [WTM+20] (AG). Tufano *et al.* [TPW+19] do not report results for $K > 1$ for the MG task. Thus, we only compute $K = 1$.

BLEU score (Bilingual Evaluation Understudy) [PRWZ02] measures how similar the candidate (predicted) and reference (oracle) texts are. Given a size n, the candidate and reference texts are broken into n-grams and the algorithm determines how many n-grams of the candidate text appear in the reference text. The BLEU score ranges between 0 (the sequences are completely different) and 1 (the sequences are identical). We use different BLEU-n scores, depending on the ones used in the reference paper of the baseline (see Table 3.3). For the CS task, we report BLEU-{1, 2, 3, 4} and their geometric mean (*i.e.,* BLEU-A); for the MG task we only report BLEU-A.

ROUGE (Recall-Oriented Understudy for Gisting Evaluation) is a set of metrics for evaluating both automatic summarization of texts and machine translation techniques in general [Lin04]. ROUGE metrics compare an automatically generated summary or translation with a set of reference summaries (typically, human-produced). We use the ROUGE LCS metrics based on the Longest Common Subsequence for the CS task [HLWM20]. Given two token sequences, X and Y, and their respective length, m and n, it is possible to compute three ROUGE LCS metrics: R (recall), computed as $\frac{LCS(X,Y)}{m}$, P (precision), computed as $\frac{LCS(X,Y)}{n}$, and F (F-measure), computed as the harmonic mean of P and R.

The computed metrics are used to select what the best training strategy for the T5 is (*i.e., no pre-training*, *pre-training single task*, or *pre-training multi-task*). We also statistically compare the performance of these three strategies for each task using the McNemar's test [McN47], which is a proportion test suitable to pairwise compare dichotomous results of two different treatments. We statistically compare each pair of training strategy in our study (*i.e., no pre-training vs pre-training single task*, *no pre-training vs pre-training multi-task*, *pre-training single task vs pre-training multi-task*) in terms of their Accuracy@1 (*i.e.,* perfect predictions) for each of the four experimented tasks. To compute the test results for two training strategies T_1 and T_2, we create a confusion matrix counting the number of cases in which (i) both T_1 and T_2 provide a correct prediction, (ii) only T_1 provides a correct prediction, (iii) only T_2 provides a correct prediction, and (iv) neither T_1 nor T_2 provide a correct prediction. We complement the McNemar's test with the Odds Ratio (OR) effect size. Also, since we performed multiple comparisons, we adjusted the obtained p-values using the Holm's correction [Hol79].

The best model output of this analysis has then been used to compare the best T5 model with the four baselines by using the performance metrics reported in Table 3.3. Moreover, we also statistically compare the Accuracy@1 of the T5 and of the baselines using the same procedure previously described (*i.e.,* McNemar's test with the OR effect size). We also perform a complementarity analysis: We define the sets of perfect predictions generated by the T5 (PP_{T5_d}) and by the baseline (PP_{BL_d}) with a beam size $K = 1$. Then, for each task and

dataset we compute three metrics:

$$Shared_d = \frac{|PP_{T5_d} \cap PP_{BL_d}|}{|PP_{T5_d} \cup PP_{BL_d}|}$$

$$OnlyT5_d = \frac{|PP_{T5_d} \setminus PP_{BL_d}|}{|PP_{T5_d} \cup PP_{BL_d}|} \qquad OnlyBL_d = \frac{|PP_{BL_d} \setminus PP_{T5_d}|}{|PP_{T5_d} \cup PP_{BL_d}|}$$

$Shared_d$ measures the percentage of perfect predictions shared between the two compared approaches on the dataset d, while $OnlyT5_d$ and $OnlyBL_d$ measure the percentage of cases in which the perfect prediction is only generated by T5 or the baseline, respectively, on the dataset d.

We also present an "inference time" analysis: we compute the time needed to run T5 on a given input. We run such an experiment on a laptop equipped with a 2.3GHz 8-core 9th-generation Intel Core i9 and 16 GB of RAM, using the CPU to run the DL model. We do this for different beam search sizes, with $K \in \{1, 5, 10, 25, 50\}$. For each K, we report the average inference time (in seconds) on all the instances of each task. Besides that, we also report the training time (in hours) for the nine different models involved in our study, *i.e., no pre-training* (four models, one for each task), *pre-training single task* (+4 models), and *pre-training multi-task* (one model pre-trained and fine-tuned for all four tasks). For the training we used a 2x2 TPU topology (8 cores) from Google Colab with a batch size of 128, with a sequence length (for both inputs and targets) of 512 tokens.

Finally, we discuss qualitative examples of predictions generated by T5 and by the baselines to give a better idea to the reader about the capabilities of these models in supporting the four code-related tasks.

3.2.2 Hyperparameter Tuning

Before running the T5 models on the test sets, we performed a hyperparameter tuning on the evaluation sets from Table 3.2, to decide the best configuration to run. This was done for all nine models we built (*e.g.,* with/without pre-training, with/without multi-task learning).

For the *pre-training* phase, we use the default parameters defined for the T5 model [RSR$^+$20]. Such a phase, indeed, is task-agnostic, and hyperparameter tuning would provide limited benefits. Instead, we tried different learning rate strategies for the *fine-tuning* phase. Especially, we tested four different learning rates: (i) *Constant Learning Rate* (C-LR): the learning rate is fixed during the whole training; (ii) *Inverse Square Root Learning Rate* (ISR-LR): the learning rate decays as the inverse square root of the training step; (iii) *Slanted Triangular Learning Rate [HR18]* (ST-LR): the learning rate first linearly increases and then linearly decays to the starting learning rate; (iv) *Polynomial Decay Learning Rate* (PD-LR): the learning rate decays polynomially from an initial value to an ending value in the given decay steps. Table 3.4 reports the specific parameters we use for each scheduling strategy.

In total, we fine-tuned 36 models (*i.e.,* nine models with four different schedulers) for 100k steps each. To select the best configuration for each training strategy, we compute the following metrics: for BF and AG, we compute the percentage of perfect predictions achieved on the evaluation set with the greedy decoding strategy (Accuracy@1); for MG, we compute

Table 3.4. Learning-rates tested for hyperparameter tuning.

Learning Rate Type	**Parameters**
Constant	$LR = 0.001$
Inverse Square Root	$LR_{starting} = 0.01$ $Warmup = 10,000$
Slanted Triangular	$LR_{starting} = 0.001$ $LR_{max} = 0.01$ $Ratio = 32$ $Cut = 0.1$
Polynomial Decay	$LR_{starting} = 0.01$ $LR_{end} = 0.001$ $Power = 0.5$

Table 3.5. Overall results achieved by the T5 model for each tasks. The best configuration is highlighted in bold

Task	**Dataset**	**Model Configuration**	**Accuracy@1**	**Accuracy@5**	**Accuracy@10**	**Accuracy@25**	**Accuracy@50**	**BLEU-A**
Automatic Bug-Fixing	BF_{small}	*no pre-training*	**16.70%**	29.88%	34.37%	39.57%	42.86%	-
		pre-training single task	15.08%	32.08%	37.01%	42.51%	45.94%	-
		pre-training multi-task	11.61%	**35.64%**	**43.87%**	**52.88%**	**57.70%**	-
	BF_{medium}	*no pre-training*	10.50%	17.60%	20.53%	24.38%	27.62%	-
		pre-training single task	**11.85%**	**19.41%**	23.28%	28.60%	32.43%	-
		pre-training multi-task	3.65%	19.17%	**24.66%**	**30.52%**	**35.56%**	-
Injection of Code Mutants	MG_{ident}	*no pre-training*	25.78%	-	-	-	-	78.26%
		pre-training single task	28.72%	-	-	-	-	**78.69%**
		pre-training multi-task	**28.92%**	-	-	-	-	78.29%
Generation of Asserts in Test	AG_{raw}	*no pre-training*	60.95%	59.14%	62.41%	69.05%	71.97%	-
		pre-training single task	**68.93%**	**75.95%**	**77.70%**	**79.24%**	**80.22%**	-
		pre-training multi-task	58.60%	66.90%	70.31%	73.19%	74.58%	-
	AG_{abs}	*no pre-training*	47.81%	49.60%	55.04%	64.28%	68.57%	-
		pre-training single task	**56.11%**	**71.26%**	**74.32%**	**76.67%**	**78.02%**	-
		pre-training multi-task	44.90%	63.40%	68.23%	73.04%	73.12%	-
Code Summarization	*CS*	*no pre-training*	11.80%	-	-	-	-	24.67%
		pre-training single task	**12.02%**	-	-	-	-	**25.21%**
		pre-training multi-task	11.45%	-	-	-	-	24.90%

the BLEU score [PRWZ02]; for CS, we compute BLEU-A, the geometric average of the BLEU-{1,2,3,4} scores [PRWZ02]. Basically, for each task we adopt one of the evaluation metrics used in the original paper. The complete results of the hyperparameters tuning phase are reported in our replication package [repg].

3.3 Results Discussion

We discuss our results accordingly to the formulated RQs.

3.3.1 Performance of T5 (RQ_1) and impact of transfer learning on performance ($RQ_{1.1}$-$RQ_{1.2}$)

Table 3.5 reports the performance achieved by the different variants of the T5 model that we experimented with. For each task (*e.g.,* Automatic Bug-Fixing) and for each dataset (*e.g.,* BFsmall), performance metrics are reported for the three adopted training strategies

(*i.e.,* no pre-training, pre-training single task, and pre-training multi-task). For readability reasons, we only report the BLEU-A, but the results of the other BLEU scores (*e.g.,* BLEU-4) are available in our online appendix [repg].

Table 3.6 reports the results of the statistical analysis we performed using the McNemar's test [McN47] to identify (if any) statistical differences in terms of Accuracy@1 when using different training strategies.

Table 3.6. McNemar's test (adj. p-value and OR) considering only accuracy@1 matches as correct predictions

Task	Dataset	Model Configuration	p-value	OR
Automatic Bug-Fixing	BF_{small}	*no pre-training vs pre-training single task*	< 0.001	0.77
		no pre-training vs pre-training multi-task	< 0.001	0.46
		pre-training multi-task vs pre-training single task	< 0.001	1.67
	BF_{medium}	*no pre-training vs pre-training single task*	< 0.001	1.56
		no pre-training vs pre-training multi-task	< 0.001	0.12
		pre-training multi-task vs pre-training single task	< 0.001	8.56
Injection of Code Mutants	MG_{ident}	*no pre-training vs pre-training single task*	< 0.001	1.51
		no pre-training vs pre-training multi-task	< 0.001	1.38
		pre-training multi-task vs pre-training single task	0.75	0.99
Generation of Asserts in Test	AG_{raw}	*no pre-training vs pre-training single task*	< 0.001	3.39
		no pre-training vs pre-training multi-task	< 0.001	0.71
		pre-training multi-task vs pre-training single task	< 0.001	4.95
	AG_{abs}	*no pre-training vs pre-training single task*	< 0.001	2.55
		no pre-training vs pre-training multi-task	< 0.001	0.74
		pre-training multi-task vs pre-training single task	< 0.001	2.93
Code Summarization	*CS*	*no pre-training vs pre-training single task*	< 0.001	1.13
		no pre-training vs pre-training multi-task	< 0.001	0.83
		pre-training multi-task vs pre-training single task	< 0.001	1.40

Focusing on the Accuracy@1, it is evident that there is no training strategy being the best one across all tasks and datasets. In particular: *no pre-training* works better on the BF_{small} dataset for automatic bug-fixing; *pre-training single task* works better on the BF_{medium} dataset for automatic bug-fixing, on both datasets related to the generation fo assert statements, and for the code summarization task; finally, *pre-training multi-task* works better for the injection of code mutants. Overall, the *pre-training single task* strategy seems to be the best performing strategy. Indeed, even when it is not the first choice for a given task/dataset, it is the second best-performing training strategy. Also, by looking at Table 3.6 we can observe that:

1. When *pre-training single task* is the best strategy, its performance in terms of Accuracy@1 are significantly better (p-value < 0.001) than the second best-performing strategy, with ORs going from 1.13 (for *CS*) to 3.39 (AG_{raw}). This means that chances of getting a perfect predictions using this strategy are 13% to 339% higher when using this strategy as compared to the second choice.

2. When *pre-training single task* is not the best strategy, but the second choice, the difference in Accuracy@1 is not significant when compared to *pre-training multi-task* for

MG_{ident}. The only significant difference is the one in favor of *no pre-training* on BF_{small}, with an OR of 0.77.

For these reasons, in our RQ_2 we will compare the T5 using the *pre-training single task* strategy against the baselines.

A few observations can be made based on the findings in Table 3.5. First, the additional pre-training is, as expected, beneficial. Indeed, on five out of the six datasets the T5 performs better with pre-training. Second, the multi-task setting did not help in most of cases. Indeed, with the exception of MG_{ident} in which the performance of *pre-training single task* and *pre-training multi-task* are basically the same, the *single task* setting performs always better. Such a result, while surprising at a first sight, can be explained by diverse types of input/output handled by the models across the four tasks. Indeed, (i) the datasets related to automatic bug-fixing and AG_{abs} include abstracted code instances as input/output; (ii) the dataset used for code mutants and AG_{raw} feature raw code instances as input/output; and (iii) the one for code summarization has raw source code as input and natural language text as output. Basically, given the different formats, the transfer learning across different tasks is likely to hinder the model rather than helping it.

Differently, the pre-training dataset features all three input/output representations and, thus, provides the model with a basic knowledge about all of them that, as a result, boosts performance.

While we will discuss more in depth the performance of the T5 model when comparing it to the considered baselines (Section 3.3.2), it is also worth commenting on the ability of the T5 to generate correct predictions, namely outputs that are identical to the reference ones (*e.g.,* a method summary equal to the one manually written by developers). Quite impressive are the performances achieved on the generation of assert statements, especially on the dataset dealing with raw source code, in which the T5 correctly predicts 68.93% of assert statements with a single guess (75.95% when using five guesses). The Accuracy@1 is instead much lower for the other tasks, ranging between 11.85% (fixing bugs in the most challenging BF_{medium} dataset) up to 28.72% when injecting mutants. Also worth noticing is the 12.02% of code summaries generated by the T5 that are identical to the manually written ones. In the next subsection, together with a comparison of our model with the baselines, we present qualitative examples of predictions generated by the T5.

3.3.2 Competitiveness of the T5 model compared to the baselines (RQ_2)

We compare the results achieved by the T5 model when using the *pre-training single task* strategy with the baseline we consider for each task (Table 3.3). The comparison is depicted in Fig. 3.2, while Table 3.8 shows the results of the statistical tests, and Table 3.10 shows the overlap metrics described in Section 3.2.1.

3.3.2.1 Automatic Bug Fixing (BF)

When using T5 for automatically fixing bugs, the accuracy achieved using a greedy decoding strategy ($K = 1$) differs according to the dataset we consider. For example, the T5 model

Delete						
	BF_{small}			BF_{medium}		
	Oracle	**Baseline [TWB$^+$19a]**	**T5**	**Oracle**	**Baseline [TWB$^+$19a]**	**T5**
Delete TypeAccess at Invocation	2,016	402	450	1,926	125	250
Delete Invocation at Block	1,444	294	326	1,315	159	240
Delete TypeAccess at ThisAccess	821	92	134	598	32	81
Delete VariableRead at Invocation	818	51	106	1,106	61	126
Delete FieldRead at BinaryOperator	479	92	100	651	66	116
Insert						
	BF_{small}			BF_{medium}		
	Oracle	**Baseline [TWB$^+$19a]**	**T5**	**Oracle**	**Baseline [TWB$^+$19a]**	**T5**
Insert Block at If	486	3	28	828	3	48
Insert Literal at BinaryOperator	468	5	27	736	0	37
Insert If at Block	406	2	22	659	0	33
Insert BinaryOperator at If	380	3	23	634	0	36
Insert VariableRead at Invocation	328	10	33	675	0	38
Move						
	BF_{small}			BF_{medium}		
	Oracle	**Baseline [TWB$^+$19a]**	**T5**	**Oracle**	**Baseline [TWB$^+$19a]**	**T5**
Move Invocation from Block to Invocation	633	17	61	1,005	4	86
Move VariableRead from Invocation to VariableRead	158	7	11	281	2	19
Move Assignment from Block to Assignment	120	0	13	209	1	19
Move Invocation from BinaryOperator to Invocation	95	7	11	183	1	14
Move BinaryOperator from BinaryOperator to BinaryOperator	68	0	2	174	0	9
Update						
	BF_{small}			BF_{medium}		
	Oracle	**Baseline [TWB$^+$19a]**	**T5**	**Oracle**	**Baseline [TWB$^+$19a]**	**T5**
Update Wra at Method	280	15	37	191	1	22
Update TypeAccess at Invocation	201	17	41	404	18	115
Update Invocation at Block	115	0	8	153	2	21
Update VariableRead at Invocation	101	1	12	226	0	19
Update BinaryOperator at If	56	3	8	148	1	12

Table 3.7. Top-20 AST operations needed to fix bugs in our dataset (see "Oracle" column) and their presence in correct predictions generated by T5 and the baseline

achieves 15% of perfect predictions on the BF_{small} dataset against 9% achieved by the baseline, with an improvement of 6 percentage points, while in the most challenging scenario, (*i.e.,* BF_{medium}) our model obtains an improvement of 8 percentage points over the baseline (11% *vs* 3%). Such improvements are statistically significant (Table 3.8) with ORs of 2.39 (BF_{small}) and 6.88 (BF_{medium}), indicating higher chance of observing a perfect prediction when using the T5 as compared to the baseline. Worth noticing is that as the beam width increases, the performance of the T5 and of the baseline gets closer, with the baseline performing better for $K = 25$ and $K = 50$ on BF_{small}.

Looking at the overlap metrics (Table 3.10), 25.90% of perfect predictions on BF_{small} and 28.78% on BF_{medium} are shared by the two techniques. The remaining are perfect predictions only with T5 (53.20% on BF_{small} and 36% on BF_{medium}) or only with the baseline (20.90% on BF_{small} and 35.16% on BF_{medium}). This indicates that the two approaches are complementary for the bug fixing task suggesting that further improvements could be possible by exploiting customized ML-based bug-fixing techniques. To further look into this finding, we analyzed the type of "code transformation" that T5 and the baseline were able to learn. With

Table 3.8. McNemer's test considering the correct predictions achieved by the T5 model and the baselines when both techniques generate only one prediction (*i.e., accuracy@1*)

Task	*Dataset* (d)	***p*-value**	**OR**
Automatic Bug-Fixing	BF_{small}	< 0.001	2.39
	BF_{medium}	< 0.001	6.88
Injection of Code Mutants	MG_{ident}	< 0.001	2.95
Generation of Asserts in Test	AG_{abs}	< 0.001	6.19
	AG_{raw}	< 0.001	43.12
Code Summarization	CS	< 0.001	35.56

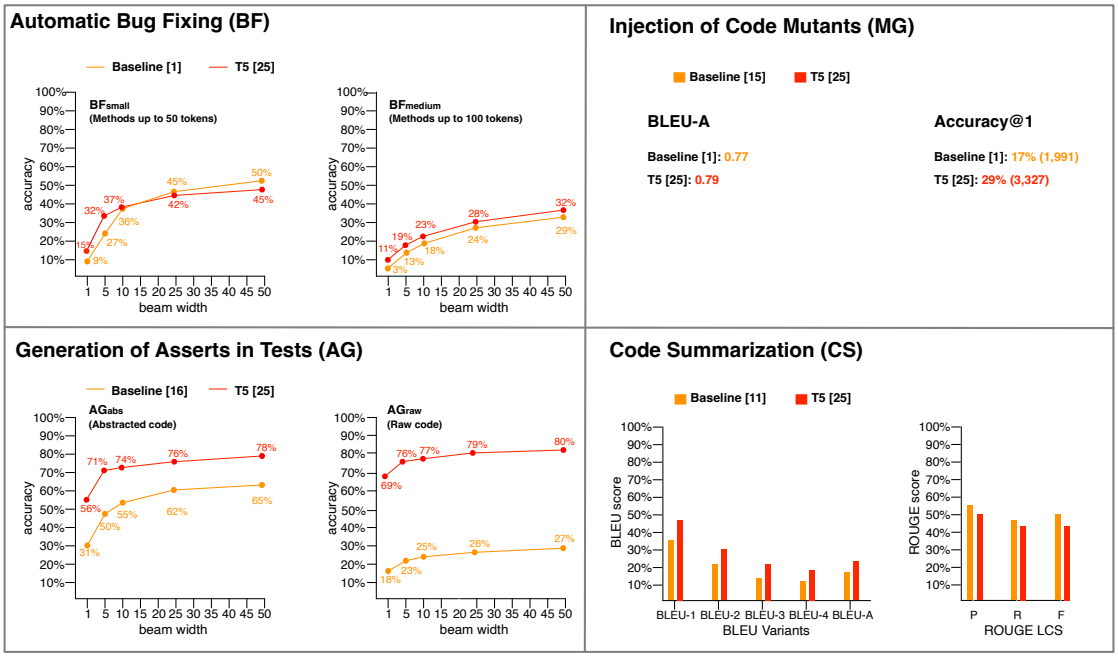
"code transformation" we refer to Abstract Syntax Tree (AST) operations needed to correctly transform the input code into the target prediction (*i.e.,* the AST operations performed by developers to transform the buggy code into the fixed code). In particular, we used the Gumtree Spoon AST Diff [FMB+14] to collect the *Delete*, *Insert*, *Move* and *Update* operations performed on the AST nodes when fixing bugs. Then, for each of these operations, we extracted the 5 most popular ones (*e.g.,* the five most popular *Delete* node operations). These 20 AST-level operations (4 types of operations × 5 most popular for each type) characterize the successful fixing of bugs/injection of code mutants in the three datasets. The column "Oracle" of (Table 3.7) reports such numbers. Then, we took the correct predictions generated by T5 and by the baselines and checked the extent to which those predictions feature the "popular" AST operations that, accordingly to our oracles, are needed to properly fix bugs. Table 3.7 reports for both techniques and both datasets (BF_{small} and BF_{medium}) the number of times the different AST operations were performed by the models.

Given the previously discussed superior performance of T5, it is expected to see that it managed to correctly perform the needed AST operations more often than the baseline. However, what is interesting is that there are specific types of operations that are not learned by the baseline while they are successfully implemented by T5. This is especially true for less popular operations such as the *Insert* ones, that require to synthesize new nodes that were not present in the input AST. In BF_{medium}, four of the top-five AST Insert operations are never applied by the baseline (see Table 3.7). Similar results are also obtained for the *Update* operations, while both models work similarly well when the bug-fix mostly requires the deletion of existing AST nodes.

3.3.2.2 Injection of Code Mutants (MG)

Looking at Fig. 3.2 we can observe that using T5 to generate mutants allows to obtain more accurate results than the baseline, with the Accuracy@1 improving by 12 percentage points, with 1,336 additional perfect predictions. The average BLEU score also improves by ∼0.02 on top of the very good results already obtained by the baseline (*i.e.,* 0.77). Minor improvements in BLEU score can still indicate major advances in the quality of the generated solutions [CL20]. Also in this case differences in terms of Accuracy@1 are statistically significant, with the T5 model being more likely to generate correct solutions ($OR = 2.95$) as

Figure 3.2. Performance of the T5 model against the experimented baselines.



compared to the baseline approach [TWB+19b] (Table 3.8).

Differently from the bug-fixing task, for the injection of code mutants the percentage of shared perfect predictions (Table 3.10) is slightly higher (33%) with, however, T5 being the only one generating 50.52% of perfect predictions as compared to the 16.48% generated exclusively by the baseline.

Similarly to what has been done in the context of the bug-fixing task, Table 3.9 reports the top-20 AST-level operations needed to correctly inject mutants in our dataset. Note that, differently from what observed for the bug-fixing task, injecting mutants mostly requires the insertion of new AST nodes. The trend that we observe is, as expected, the opposite of what we found for the bug-fixing task because the task is the same but with reversed input/output. Indeed, the baseline seems to correctly predict the most popular *Insert* operations in the AST, while it almost ignores the more rare *Delete* ones. T5 instead, covers all top-20 operations.

3.3.2.3 Generation of Assertions in Test Methods (AG)

T5 achieve much better performance in this task as compared to the baseline. The gap is substantial both with (AG_{abs}) and without (AG_{raw}) code abstraction (Fig. 3.2). With abstraction, the T5 achieves a 56% accuracy at $K = 1$ against the 31% achieved by *ATLAS* [WTM+20]. When both approaches are asked to generate multiple assert statements (*i.e.,* $K = 5, 10, 25, 50$) the gap in performance ranges between 13-25 percentage points. When using the more challenging non-abstracted dataset AG_{raw}, T5 achieves even better results.

Table 3.9. Top-20 AST operations needed to inject mutants in our dataset (see "Oracle" column) and their presence in correct predictions generated by T5 and the baseline

Delete			
	MG_{ident}		
	Oracle	**Baseline [TWB$^+$19b]**	**T5**
Delete TypeAccess at Invocation	387	1	30
Delete Return at Block	327	20	64
Delete FieldRead at BinaryOperator	283	0	7
Delete FieldRead at Invocation	242	0	19
Delete Invocation at Block	236	0	15
Insert			
	MG_{ident}		
	Oracle	**Baseline [TWB$^+$19b]**	**T5**
Insert TypeAccess at Invocation	6,230	1,125	1,744
Insert Invocation at Block	3,979	860	1,183
Insert TypeAccess at ThisAccess	2,219	479	722
Insert VariableRead at Invocation	2,061	245	466
Insert Block at If	1,795	485	671
Move			
	MG_{ident}		
	Oracle	**Baseline [TWB$^+$19b]**	**T5**
Move Invocation from Block to Invocation	1,154	225	356
Move Invocation from Return to Invocation	283	55	105
Move Return from Block to Return	224	58	100
Move Assignment from Block to Assignment	190	26	56
Move Invocation from Invocation to Invocation	129	1	27
Update			
	MG_{ident}		
	Oracle	**Baseline [TWB$^+$19b]**	**T5**
Update TypeAccess at Invocation	923	67	220
Update FieldRead at BinaryOperator	408	14	63
Update Wra at Method	264	1	31
Update TypeAccess at ThisAccess	228	10	73
Update TypeReference at Method	208	0	25

In this regard, when T5 is asked to generate only one assert statement ($K = 1$), the reported accuracy is 51 percentage points higher compared to the baseline , while for larger K values the gap in performance ranges between 51-53 percentage points. The McNemar's test confirms the huge gap in performance between the two techniques, with *ORs* ranging between 6.19 (AG_{abs}) and 43.12 (AG_{raw}).

In terms of overlap, we found a trend similar to the previously discussed task (mutants injection): On AG_{abs} we have 34.92% of perfect predictions shared between the two approaches, while the remaining instances are distributed between the ones only predicted by T5 (58.87%) and the ones only predicted by the baseline (6.21%). The overlap is much smaller on the AG_{raw} dataset, with only 9.56% of the instances correctly predicted by both the approaches, 89.65% of them correctly predicted only by T5, and 0.79% only by the baseline.

3.3.2.4 Code Summarization (CS)

On this task, T5 achieves a substantial increase in BLEU score as compared to the baseline. When considering the average BLEU (BLEU-A), the improvement is of ∼5 percentage points. On the other hand, it can be noticed that the ROUGE-LCS scores achieved when using T5 are lower than the ones achieved by the baseline (∼5 percentage points lower on the F-measure score). Thus, looking at these metrics, there is no clear winner, but T5 seems to be at least comparable to the baseline. To have something easier to interpret, we compared the two approaches in terms of the number of perfect predictions they generate, despite the fact that such a metric was not used in the original paper [HLWM20]. This means counting the comments generated by a technique that are exactly equal to the ones manually written by humans. T5 managed to generate 12.02% of perfect predictions (10,929 instances) against the 3.4% (3,048) of the baseline technique (over 3 × better). As expected from previous results, the majority of the perfect predictions for this task can be done only using T5 (93.79%). A limited percentage of perfect predictions is shared (4.79%), and a minority of instances can be only predicted through the baseline (1.42%). The McNemar's test highlights a statistical significance in terms of Accuracy@1, with an OR of 35.56.

Table 3.10. Overlap metrics for correct predictions generated by the T5 model and the baselines.

Task	*Dataset* (d)	$Shared_d$	$OnlyT5_d$	$OnlyBL_d$
Automatic Bug-Fixing	BF_{small}	25.90%	53.20%	20.90%
	BF_{medium}	28.78%	36.06%	35.16%
Injection of Code Mutants	MG_{ident}	33.00%	50.52%	16.48%
Generation of Asserts in Test	AG_{abs}	34.92%	58.87%	6.21%
	AG_{raw}	9.56%	89.65%	0.79%
Code Summarization	*CS*	4.79%	93.79%	1.42%

3.3.3 Qualitative Analysis

To give a better idea to the reader about the capabilities of the T5 model in supporting the four code-related tasks, Fig. 3.3 shows two examples of perfect predictions made by T5 for each task. Each example is bordered with a dashed line and shows (i) the input provided by the model, and (ii) the generated output. In particular, in the case of the bug-fixing, mutants injection, and code summarization tasks, the first line shows the input and the second the output. Concerning the generation of assert statements, the first two lines (*i.e.,*

Figure 3.3. Examples of perfect and alternative predictions

those marked with "//Test method" and "//Focal method") represent the input, while the third line shows the generated assert statement. We highlighted in bold the most relevant parts of the output generated by the model. The bottom part of Fig. 3.3 also shows some "wrong" predictions (*i.e.,* the output of the model is different from the expected target) for the code summarization task, that we will discuss later on.

Concerning the bug-fixing task, in the first example the model adds the `break` statement to each `case` of the `switch` block, thus allowing the program to break out of the `switch` block after one `case` block is executed. In the second example, instead, it changes the execution

order of a statement as done by developers to fix the bug.

As per the mutants injection, the first example represents an *arithmetic operator deletion*, while the second is a *non void method call mutation* [pit]. While these transformations might look trivial, it is worth remembering that they are considered as correct since they reproduce real bugs that used to affect these methods. Thus, the model is basically choosing where to mutate and what to mutate in such a way to simulate real bugs (accomplishing one of the main goals of mutation testing).

Both examples of correct prediction we report involve the generation of an assert statement including an invocation to the focal method (*i.e.,* the main method tested by the test method). While the first is a rather "simple" `assertFalse` statement, the second required the guessing of the expected value (*i.e.,* `assertEquals`).

Finally, for the code summarization, the two reported examples showcase the ability of T5 to generate meaningful summaries equivalent to the ones manually written by developers. For this task, we also reported in the bottom part of the figure some wrong but still meaningful predictions. In this case, the grey text represent the original summary written by developers, while the bold one has been generated by T5. In both cases, the generated summary is semantically equivalent and even more detailed that the manually written one.

These two examples help in discussing an important limitation of our analysis: While we assume the correct predictions to be the *only* valuable outputs of T5 and of the experimented baselines, they actually represent a lower-bound for their performance. Indeed, there are other predictions that, even if wrong, could still be valuable for developers, such as the two shown for the code summarization task.

3.3.4 Training and Inference Time

Table 3.11 reports the training time (in hours) for the nine models we trained. On average, the infrastructure we used for training requires 31.5 seconds every 100 training steps which, given our batch size = 128, means that 12,800 training instances can be processed in 31.5 seconds. Of course, multiple passes (usually referred to as epochs) are needed on the dataset during the training. Table 3.11 shows that (i) the pre-training has a cost of ∼22h that should be added on top of the fine-tuning cost shown for each task; (ii) as expected, the training time increases with the increase in size of the training dataset, with the *code summarization* task being the most expensive in terms of training time; (iii) clearly, the *multi-task* setting requires to train the model on all tasks, resulting in the highest training time (175h).

Table 3.11. Training time (hours) for the trained T5 models

Training	Bug-fixing	Mutants generation	Generation of assert statements	Code summarization	Multi-Task
No pre-training	6.26	5.85	17.51	123.55	-
Pre-training	28.10	27.72	39.40	145.42	175.00

Table 3.12 presents, instead, the results of the inference time analysis (*i.e.,* the time needed to run the model on a given input and obtain the prediction). Such analysis allows

to understand the extent to which such a model can be used in practice. Table 3.12 reports the inference time in seconds for different K values (*e.g.,* with $K = 10$ the reported time is the one required by the model to generate 10 possible solutions).

Table 3.12. Inference time with different beam size values.

K	BF_{small}	BF_{medium}	MG_{ident}	AG_{abs}	AG_{raw}	CS
1	0.72	1.86	0.94	0.73	0.53	0.20
5	1.47	3.69	1.70	1.59	1.04	0.36
10	1.91	5.26	2.20	2.64	1.52	0.48
25	3.54	11.10	4.32	5.45	3.15	0.81
50	5.99	20.90	7.60	10.24	5.45	1.45

Concerning the bug-fixing task, the time needed to generate a fix depends on the dataset, since the complexity of the instances they feature is different. In the BF_{small} dataset, the average inference time ranges between 0.72s ($K = 1$) and 5.99s ($K = 50$), while it is larger on the BF_{medium} dataset (1.86s for $K = 1$ and 20.90s for $K = 50$). For the injection of code mutants, we observed results comparable to those of BF_{small}: with $K = 1$ the average inference time is 0.94s, while for $K = 50$ it is 7.60s. The generation of assert statement is very fast for low values of K (0.73s for AG_{abs} and 0.53s for AG_{raw} with $K = 1$), while it gets slower for higher values of K (10.24 for AG_{abs} and 5.45 for AG_{raw} with $K = 50$). Finally, concerning the code summarization task, T5 takes only 0.20s for $K = 1$ and 1.45s for $K = 50$ to output code summaries for a method given as input.

Overall, considering that all the targeted tasks do not have strong real-time constraints (*e.g.,* a developer can wait a few seconds for the automated fixing of a bug), the inference times should not hinder the model applicability in practice. Also, the reported inference times were obtained by running the model on a consumer-level device and by only using CPUs. We also computed the inference time using an Nvidia Tesla P100 GPU equipped with 16GB of VRAM. The achieved results are available in our replication package [repg]. In summary, we observed an average decrease of inference time of ∼70% as compared to the one obtained using the CPU.

3.4 Threats to Validity

Construct validity. Threats to construct validity concern the relationship between theory and observation. We used existing datasets that are popular and used in the community for both pre-training and fine-tuning our model with minimal additional processing (*e.g.,* removal of duplicates after abstraction in the dataset used for the pre-training). Additionally, we have released all of our code and models in our replication study [repg] for verification.

Internal validity. Threats to internal validity concern factors, internal to our study, that could influence its results. Many factors can influence our results, from model architecture, hyperparameter choices, data processing, the data itself, etc. For mitigating these issues, we have adopted methodologies usually employed in DL-based research. Specifically, we per-

formed a detailed analysis of hyperparameter choices as discussed in Section 3.2.2. Concerning the pre-training phase, we used the default T5 parameters selected in the original paper [RSR+20] since we expect little margin of improvement for such a task-agnostic phase. For the fine-tuning, due to computational feasibility reasons, we did not change the model architecture (*e.g.,* number of layers), but we experiment with different learning rates schedulers. We are aware that a more extensive calibration would likely produce better results. Finally, we pre-trained the model by masking 15% of tokens (*i.e.,* words in comments and code tokens in the raw and abstracted code) in the ∼2.7M instances from the pre-training dataset. However, we did not experiment with the model after pre-training to verify whether it actually learned the languages of interest (*i.e.,* raw source code, abstracted source code, and technical natural language). To address this limitation, we randomly selected 3k instances from the BF_{medium} test set, both in their abstract and raw representation (6k in total). We also selected 3k code summaries from the *CS* dataset obtaining a dataset of 9k instances, equally split across raw source code, abstracted source code, and technical natural language. Note that these are instances that have not been used to pre-train the model and, thus, are unseen for a model only subject to pre-training. We randomly masked 15% of tokens in each of those instances, asking the pre-trained model to predict them. T5 correctly predicted 87,139 out of the 102,711 masked tokens (*i.e.,* 84.8% accuracy). As expected, given the different complexity of the three "languages", T5 achieved a higher accuracy of 90.8% when working on abstracted code, 82.7% on raw code, and 64.6% when guessing tokens from technical language. Overall, such results indicate that the model successfully gathered knowledge about the languages of interest during the pre-training.

Also the quality of the employed datasets can dramatically impact the achieved results. This is because there may be biases making the dataset not representative of the real world. To assess the quality of our datasets we conducted various analyses around sampling bias and data snooping as recommended by Watson *et al.* [WCP+22]. To this end, we conducted an exploratory data analysis (EDA), which helps answering questions related to the reliability and quality of our datasets. To accomplish this, we performed a two-fold statistical procedure: complexity size and token distributions. In the complexity size procedure, we count the number of tokens per dataset and data partition. Then, we present the relative distribution in log scale. While in the token procedure, we concentrated on counting specific tokens by popularity or special interest (*e.g.,* if, $assert$, or $public$). The purpose of the EDA is to monitor the size of datasets and its impact in the model performance. EDA's results can be found in our web appendix [repg].

Conclusion validity. Threats to conclusion validity concern the relationship between evaluation and outcome. To this extent, we used appropriate statistical procedures, also adopting p-value adjustment when multiple tests were used within the same analysis.

External validity. Threats to external validity are related to the generalizability of our findings. Our study focused on the T5 model on four tasks using six datasets, all of which only involved Java code. While it is unclear how our model would perform if trained on other programming languages, excluding the abstraction component, the whole pipeline is language agnostic and can be easily adapted to other languages for evaluating this.

We also performed an analysis of our dataset aimed at finding out the generalizability of

our models. This analysis assessed the level of data snooping among our datasets' training and test sets and how this impacts our model's results. To accomplish this we calculate the overlap between our fine-tuning datasets' training and test sets by computing the pairwise Levenshtein Distance [Lev66] between the two sets. With these distances calculated, we computed the correlation between the distances and the performance of our model on the different test sets. Specifically, we selected a statistically representative sample (confidence level = 95% and confidence interval = 5%) of each training set and calculated the pairwise Levenshtein Distance [Lev66] between it and the entirety of the test set for each fine-tuning dataset. Next, depending on the type of performance metric (Perfect Prediction or BLEU Score), we calculate the correlation between the minimum, median, and maximum distances of all sampled training examples to each test example and the performance of our model on the test set. For the perfect prediction, we use Point Biserial Correlation (PBC) [Tat54] as it allows to compare binary and continuous data. For the BLEU score, we use Pearson Correlation [Tat54] since both are continuous values.

Table 3.13. Correlation between training-test set similarity and test set performance.

Dataset	Min	Median	Max
BF_{small}	-0.15	-0.03	0.04
BF_{medium}	-0.05	-0.03	0.01
MG_{ident}	0.21	0.03	-0.23
AG_{abs}	-0.21	-0.14	0.29
AG_{raw}	-0.21	-0.14	0.19
CS	-0.38	-0.17	-0.09

Table 3.13 shows the correlation for each dataset. As shown, there exists a negative correlation between the minimum and median distances and model performance, *i.e.,* the model tends to perform worse as the distance between the training and test examples increases. For the maximum distance case, there is instead a positive correlation for perfect prediction performance, *i.e.,* the model tends to perform better the further away the maximum training examples are from the test examples. Such a result may be simply due to specific outliers present in the test set (*i.e.,* an instances being very far from the ones in the training set). However, all the correlations we observed are quite low, supporting the generalizability of our models.

3.5 Conclusions

We presented an empirical study aimed at investigating the usage of transfer learning for code-related tasks. In particular, we pre-trained and fine-tuned several variants of the Text-To-Text Transfer Transformer model with the goal of supporting four code-related tasks, namely *automatic bug-fixing*, *injection of code mutants*, *generation of assert statements in test methods*, and *code summarization*. We compared the performance achieved by the T5 against state-of-the-art baselines that proposed DL-based solutions to these four tasks.

The achieved results showed that: (i) the pre-training process of the T5, as expected,

boosts its performance across all tasks; (ii) the multi-task fine-tuning (*i.e.,* a single model trained for different tasks) instead, does not consistently help in improving performance, possibly due to the different types of "data" manipulated in the four tasks (*i.e.,* raw code, abstracted code, natural language); (iii) in its best configuration, the T5 performs better that the baselines across all four tasks. When looking at the latter finding it is important to remember that the baselines used for comparison are not pre-trained and, thus, they (i) exploited less training data, and (ii) did not need the additional ∼22 hours of computation required by the pre-training.

4

Evaluating the Robustness of DL-based techniques for Generating Code

One of the long lasting dreams in software engineering research is the automated generation of source code. Towards this goal, numerous approaches have been introduced (see Chapter 2). The release of *GitHub Copilot* [CTJ+21] pushed the capabilities of these tools to whole new levels. The large-scale training performed on the OpenAI's Codex model allows Copilot to not limit its recommendations to few code tokens/statements the developer is likely to write: Copilot is able to automatically synthesize entire functions just starting from their signature and natural language descriptions.

This new generation of code recommender systems has the potential to change the way in which developers write code [EB22] and comes with a number of questions concerning how to effectively exploit them to maximize developers' productivity. Intuitively, the ability of the developer to provide "proper" inputs to the model will become central to boost the effectiveness of its recommendations. In the concrete example of GitHub Copilot, the natural language description provided to the model to automatically generate a code function could substantially influence the model output. This means that two developers providing different natural language descriptions for the same function they would like to automatically generate could receive two different recommendations. While this would be fine in case the two descriptions are actually different in the semantics of what they describe, receiving different recommendations for *semantically equivalent natural language descriptions* would pose questions on the robustness and usability of DL-based code recommenders.

This is the main research question we investigate in this chapter: We study the extent to which different semantically equivalent natural language descriptions of a function result in different recommendations (*i.e.,* different synthesized functions) by GitHub Copilot. The latter is selected as representative of DL-based code recommenders since it is the *de facto* state-of-the-art tool when it comes to code generation.

We collected from an initial set of 1,401 open source projects a set of 892 Java methods that are (i) accompanied by a Doc Comment for the Javadoc tool, and (ii) exercised by a test suite written by the project's contributors. Then, as done in the literature [HLX+18a, LYX+20], we considered the first sentence of the Doc Comments as a "natural language

description" of the method. We refer to this sentence as the "*original*" description.

We preliminarily check whether existing automated paraphrasing techniques are suitable for robustness testing, *i.e.,* if they can be used to create semantically equivalent descriptions of the methods to generate. We validate two state-of-the-art approaches in this scenario: PEGASUS [ZZSL20a], a DL-based paraphrasing tool, and Translation Pivoting (TP), a heuristic-based approach. We use both techniques to generate a paraphrase for each *original* description in our dataset. Then, we manually inspect the obtained paraphrases and classified them as semantically equivalent or not.

To answer our main research question, we generated different paraphrases for each *original* description. We use the two previously described automated approaches, *i.e.,* PEGASUS and TP, and we also manually generated paraphrases by distributing the original descriptions among four of the authors, each of which was in charge of paraphrasing a subset of them. Therefore, for each *original* description, we obtain a set of semantically equivalent *paraphrased* descriptions. We provided both the *original* and the *paraphrased* descriptions as input to *Copilot*, asking it to generate the corresponding method body. We analyze the percentage of cases in which the *paraphrased* descriptions result in a different code prediction as compared to the *original* one, with a particular focus on the impact on the prediction quality, *e.g.,* cases in which the *original* description resulted in the recommendation of a method passing its associated test cases while switching to a *paraphrased* description made *Copilot* recommending a method failing its related tests.

Our results show that paraphrasing a description results in a change in the code recommendation in ∼46% of cases. The resulting changes also cause substantial variations in the percentage of correct predictions. Such findings indicate the central role played by the model's input in the code recommendation and the need for testing and improving the robustness of DL-based code generators.

Data and code used in our study are publicly available [repa].

4.1 Study Design

The *goal* of our study is to understand how robust is a state-of-the-art DL-based code completion approach (*i.e., GitHub Copilot*). We aim at answering the following research questions:

RQ_0: To what extent can automated paraphrasing techniques be used to test the robustness of DL-based code generators? Not always natural language processing techniques can be used out of the box on software-related text [LZB+]. Therefore, with this preliminary RQ, we want to understand whether existing automated techniques for generating natural language paraphrases are suitable for SE task at hand (*i.e.,* paraphrasing a function description).

RQ_1: To what extent is the output of GitHub Copilot influenced by the code description provided as input by the developer? This RQ aims at understanding whether *Copilot*, as a representative of DL-based code generators, is likely to generate different recommendations for different semantically equivalent natural language descriptions provided as input.

In the following we detail the context for our study (Section 4.1.1) and how we collected (Section 4.1.2) and analyzed (Section 4.1.3) the data needed to answer our RQs.

4.1.1 Context Selection

The context of our study is represented by 892 Java methods collected through the following process. We selected all GitHub Java repositories having at least 300 commits, 50 contributors, and 25 stars. These filters have been used in an attempt to exclude personal/toy projects.

We also excluded forked projects to avoid duplicates. The decision to focus on a single programming language aimed instead at simplifying the non-trivial toolchain needed to run our study. The whole repositories selection process has been performed using the GitHub search tool by Dabic *et al.* [DAB21]. At this stage, we obtained 1,401 repositories.

In our experimental design, we use the passing/failing tests as a proxy to assess the correctness of the predictions generated by *Copilot*. Thus, we need the projects to use a testing framework and to be compilable. We selected all projects that used Maven as build automation tool and for which the build of their latest release succeeded. We obtained 214 repository. By parsing the POM (Project Object Model) file[1] we only considered projects having as dependencies both jUnit [jun] — a well-known unit testing framework — and Jacoco [jac] — a code coverage library. We analyzed the Jacoco reports and selected as methods subject of our experiment those having at least 75% of statement coverage. This gives us confidence that the related test cases exercise an acceptable number of behaviors and, therefore, could allow to spot cases in which different generated functions for semantically-equivalent descriptions actually behave differently. We are aware that passing tests does not imply correctness. We discuss this aspect in Section 4.3.

Given our goal to use the method's description as input for *Copilot*, we also exclude methods not having any associated Doc Comment for the Javadoc tool. Then, we process the Doc Comment of each method in our dataset to extract from it the first sentence (*i.e.,* from the beginning to the first "."). This is the same approach used in the literature when building datasets aimed at training DL-based techniques for Java code summarization (see *e.g.,* [HLX+18a, LYX+20]), with the training set composed by pairs <`method, code_description`>, with the latter being the first sentence of the Doc Comment. To ensure that the extracted sentence contains enough wording for the code description, we exclude all methods having less than 10 tokens in the extracted first sentence, since their description may not be sufficient for synthesizing the method.

The above-described process resulted in the collection of 892 Java methods. Table 4.1 shows descriptive statistics about their characteristics in terms of number of tokens, parameters and cyclomatic complexity. These three together provide an idea about the complexity of the task *Copilot* was asked to perform (*i.e.,* the complexity of the methods it had to generate). Statistics about the coverage show, instead, the by-design high statement coverage we ensure for the included methods.

[1] POM files are used in Maven to declare dependencies towards libraries.

Full Context

```
public class Hook implements Resultsable {

    // Start: attributes from JSON file report
    private final Result result = null;
    private final Match match = null;

    @JsonDeserialize(using = OutputsDeserializer.class)
    @JsonProperty("output")
    private final Output[] outputs = new Output[0];

    // foe Ruby reports
    private final Embedding[] embeddings = new Embedding[0];
    // End: attributes from JSON file report

    @Override
    public Result getResult() {
        return result;
    }

    /** Return the embedding vector */
    public Embedding[] getEmbeddings() {
        |
    }                                          Method to be predicted

    /** Checks if the hook has content meaning as it has at least
      * attachment or result with error
      * message.
    */
    public boolean hasContent() {
        if (embeddings.length > 0) {
            return true;
        }
        if (StringUtils.isNotBlank(result.getErrorMessage())) {
            return true;
        }
        // TODO: hook with 'output' should be treated
        /  as empty or not?
        return false;
    }
}
```

Non Full Context

```
public class Hook implements Resultsable {

    // Start: attributes from JSON file report
    private final Result result = null;
    private final Match match = null;

    @JsonDeserialize(using = OutputsDeserializer.class)
    @JsonProperty("output")
    private final Output[] outputs = new Output[0];

    // foe Ruby reports
    private final Embedding[] embeddings = new Embedding[0];
    // End: attributes from JSON file report

    @Override
    public Result getResult() {
        return result;
    }

    /** Return the embedding vector */
    public Embedding[] getEmbeddings() {
        |
    }                                          Method to be predicted

}
```

Figure 4.1. *GitHub Copilot's* input for both code context representations

Table 4.1. Our dataset of 892 methods from 33 repositories

	Avg	Median	St. Dev.
# Tokens	154.3	92.0	218.2
# Parameters	1.6	1.0	1.2
# Cyclomatic Complexity	5.3	3.0	7.6
% Coverage	96.1	100.0	6.7

4.1.2 Data Collection

To address $\mathbf{RQ}_0$, we experiment with two state-of-the-art paraphrasing techniques. The first is named PEGASUS [ZZSL20a], and it is a sequence-to-sequence DL model pre-trained using self-supervised objectives specifically tailored for abstractive text summarization and fine-tuned for the task of paraphrasing [peg]. As for the second technique, we opted for Translation Pivoting (TP).

Such a technique relies on natural language translation services to translates the *original* description o from English into a foreign language (*i.e.,* French), obtaining $oE \rightarrow F$. Then, $oE \rightarrow F$ is translated back in the original language ($o_{E \rightarrow F \rightarrow E}$) obtaining a paraphrase.

We provide each technique with the *original* description as input. TP failed to generate a valid paraphrase (*i.e.,* a sentence different from the original one) in 100 cases (out of 892), while this only happened once with PEGASUS. We manually analyzed whether the valid paraphrases we obtained were actually semantically equivalent to the *original* description. For such a process, each of the 1,683 paraphrases (892 for each of the two tools minus the 101 invalid ones) has been independently inspected by two authors who classified it as semantically equivalent or not. Conflicts, that arisen in 11.9% (PEGASUS) and 16.54% (TP) of cases, have been solved by a third author not involved in the first place.

Concerning $\mathbf{RQ}_1$, we start from the *original* description and we generate semantically equivalent descriptions by (i) using the two automated tools, *i.e.,* PEGASUS [peg] and TP, and (ii) manually generating paraphrases. For the manual paraphrasing, we split the 892 methods together with their *original* description into four sets and assigned each of them to one author. Each author was in charge of writing a semantically equivalent but different description of the method by looking at its code and *original* description. This resulted in a dataset (available in [repa]) in which, for each subject method, we have its *original* and *paraphrased* description. In the end, for each *original* sentence, we had between one and three paraphrases: $paraphrased_{PEGASUS}$, $paraphrased_{TP}$, and $paraphrased_{manual}$. While $paraphrased_{manual}$ is available for all the methods, $paraphrased_{PEGASUS}$ and $paraphrased_{TP}$ are not. Indeed, we exclude the cases in which each of such tools failed to generate paraphrases (1 and 100, respectively) and the ones that were not considered as semantically equivalent in our manual check (based on the results of RQ_0). The maximum number of semantically equivalent paraphrases is 2,575 (up to 891 with PEGASUS, up to 792 with TP, and 892 manually).

The paraphrases, as well as the *original* description, have been used as input to *Copilot*, simulating developers asking it to synthesize the same Java method by using different natural language descriptions. At the time of our study, *Copilot* does not provide open APIs

to access its services. The only way to use it is through a plugin for one of the supported IDEs. Manually invoking *Copilot* for the thousands of times needed (up to 6,934, as we will explain later) was clearly not an option. For this reason, we developed a toolchain able to automatically invoke *Copilot* on the subject instances: We exploit the AppleScript language to automate this task on a MacBook Pro, simulating the developer's interaction with Visual Studio Code (*vscode*).

For each method m_i in our dataset, we created up to four different versions of the Java file containing it (one for each of the experimented descriptions). In all such versions, we (i) emptied m_i's body, just leaving the opening and closing curly bracket delimiting it; and (ii) removed the Doc Comment, replacing it with one of the four code descriptions we prepared.

Starting from these files, the automation script we implemented (available in our replication package [repa]) performs the following steps on each file F_i.

First, it opens F_i in *vscode* and moves the cursor within the curly brackets of the method m_i of interest. Then, it presses "`return`" to invoke *Copilot*, waiting up to 20 seconds for its recommendation. Finally, it stores the received recommendation, that could possibly be empty (*i.e.,* no recommendation received). To better understand this process, the top part of Fig. 4.1 depicts how the invocation of *Copilot* is performed. The gray box represents the whole Java file (*i.e.,* the context used by *Copilot* for the prediction). The emptied method (*i.e.,*`getEmbeddings`) is framed with a black border, with the cursor indicating the position in which *Copilot* is invoked. The green comment on top of the method represents one of the descriptions we created. As it can be seen, Fig. 4.1 includes for the same Java file two different scenarios, named *Full context* and *Non-full context*. In the *Full context* scenario (top part of Fig. 4.1) we provide *Copilot* with the code **preceding and following** the emptied method, simulating a developer adding a new method in an already existing Java file. In the *Non-full context* scenario, instead, we only provide as context the code preceding the emptied method (bottom part of Fig. 4.1), simulating a developer writing a Java file sequentially and implementing a new method.

The basic idea behind these two scenarios is that the contextual information provided to *Copilot* can play a role in its ability to predict the emptied method. Overall, the maximum number of *Copilot* invocations needed for our study is 6,934 (892 *original* descriptions plus up to 2,575 paraphrases, each of which for 2 context scenarios). After having collected *Copilot*'s recommendations, we found out that sometimes they did not only include the method we asked to generate, but also additional code (*e.g.,* other methods). To simplify the data analysis and to make sure we only consider one recommended method, we wrote a simple parsing tool to only extract from the generated recommendation the first valid method (if any).

4.1.3 Data Analysis

Concerning $\mathbf{RQ}_0$, we report the number and the percentage of 892 methods for which automatically generated paraphrases (*i.e.,* those generated by PEGASUS and by TP) have been classified as semantically equivalent to the *original* description. This provides an idea of how reliable these tools are when used for testing the robustness of DL-based code generators.

Also, this analysis allows to exclude from RQ_1 automatically generated paraphrases that are not semantically equivalent.

To answer $\mathbf{RQ_1}$, we preliminarily assess how far the paraphrased descriptions are from the original ones (*i.e.,* the percentage of changed words) by computing the normalized token-level Levenshtein distance [Lev66] (NTLev) between the *original* (d_o) and any *paraphrased* description (d_p):

$$NTLev(d_o, d_p) = \frac{TLev(d_o, d_p)}{\max(\{|d_o|, |d_p|\})}$$

with *TLev* representing the token-level Levenshtein distance between the two descriptions.

While the original Levenshtein distance works at character-level, it can be easily generalized at token-level (each unique token is represented as a specific character). In this case, a token is a word in the text. The normalized token-level Levenshtein distance provides an indication of the percentage of words that must be changed in the *original* description to obtain a *paraphrased* one.

Then, we analyze the percentage of methods for which the *paraphrased* descriptions result in a different method prediction as compared to the *original* one. When they are different, we also assess how far the methods obtained by using a given *paraphrased* description is from the method recommended when providing the *original* description as input. Also in this case we use the token-level Levenshtein distance as metric. The latter is computed with the same formula previously reported for the natural text descriptions; in this case, however, the tokens are not the words but the Java syntactic tokens. Thus, NTLev indicates in this case the percentage of code tokens that must be changed to convert the method obtained through the *original* description into the one recommended with one of the paraphrases.

Finally, we study the "quality" of the recommendations obtained using the different descriptions both in the *Full context* and *Non-full context* scenarios. Given the sets of methods generated from the *original* description and each of the paraphrasing approach considered, we present the percentages of methods for which *Copilot*: (i) synthesized a method passing all the related test cases (*PASS*); (ii) synthesized a method that does not pass at least one of the test cases (*FAIL*); (iii) generated an invalid method (*i.e.,* with syntactic errors) (*ERROR*); (iv) did not generate any method (*EMPTY*). Syntactic errors have been identified as recommendations for which *Java Parser* [jav] did not manage to identify a valid recommended method (*i.e.,* cases in which *Java Parser* fails to identify a method node in the AST generated for the obtained recommendation). On top of the passing/failing methods, we also compute the token-level Levenshtein distance and the CodeBLEU [RGL+20] between each synthesized method and the target one (*i.e.,* the one originally implemented by the developers). CodeBLEU measures how similar two methods are. Differently from the BLEU score [PRWZ02], CodeBLEU evaluates the predicted code considering not only the overlapping *n*-grams but also syntactic and semantic match of the two pieces of code (predicted and reference) [RGL+20].

4.1.4 Replication Package

The code and data used in our study are publicly available [repa]. In particular, we provide (i) the dataset of manually defined and automatically generated paraphrases; (ii) the AppleScript code used to automate the *Copilot* triggering; (iii) the code used to compute the CodeBLEU and the Levenshtein distance; (iv) the dataset of 892 methods and related tests used in our study; (v) the scripts used to automatically generate the paraphrased descriptions using PEGASUS and TP; and (vi) all raw data output of our experiments.

4.2 Results Discussion

As previously explained, in RQ_1 we conducted our experiments both in the *Full context* and in the *Non-full context* scenario. Since the obtained findings are similar, we only discuss in the thesis the results achieved in the *Full context* scenario (*i.e.,* the case in which we provide *Copilot* with all code preceding and following the method object of the prediction). The results achieved in the *Non-full context* scenario are available in our replication package [repa].

4.2.1 RQ_0: Evaluation of Automated Praphrase Generators

Table 4.2. Number of semantically equivalent or nonequivalent paraphrased descriptions obtained using PEGASUS and TP.

	Equivalent	Nonequivalent	Invalid
PEGASUS	666 (74.7%)	225 (25.2%)	1 (0.1%)
TP	688 (77.1%)	104 (11.7%)	100 (11.2%)

Table 4.2 reports the number of semantically equivalent and nonequivalent descriptions obtained using the two state-of-the-art paraphrasing techniques, namely PEGASUS and Translation Pivoting (TP), together with the number of invalid paraphrases generated. Out of the 892 *original* descriptions on which they have been run, PEGASUS generated 666 (75%) semantically equivalent descriptions, while TP went up to 688 (77%). If we do not consider the invalid paraphrases, *i.e.,* the cases for which the techniques do not actually provide any paraphrase, the latter obtains ∼87% of correctly generated paraphrases.

These findings suggest that the two paraphrasing techniques can be adopted as testing tools to assess the robustness of DL-based code recommenders. In particular, once established a reference description (*e.g.,* the *original* description in our study), these tools can be applied to paraphrase it and verify whether, using the reference and the paraphrased descriptions, the code recommenders generate different predictions.

Answer to RQ_0. State-of-the-art paraphrasing techniques can be used as starting point to test the robustness of DL-based code recommenders, since they are able to generate semantically equivalent descriptions of a reference text in up to 77% of cases.

4.2.2 RQ_1: Robustness of GitHub Copilot

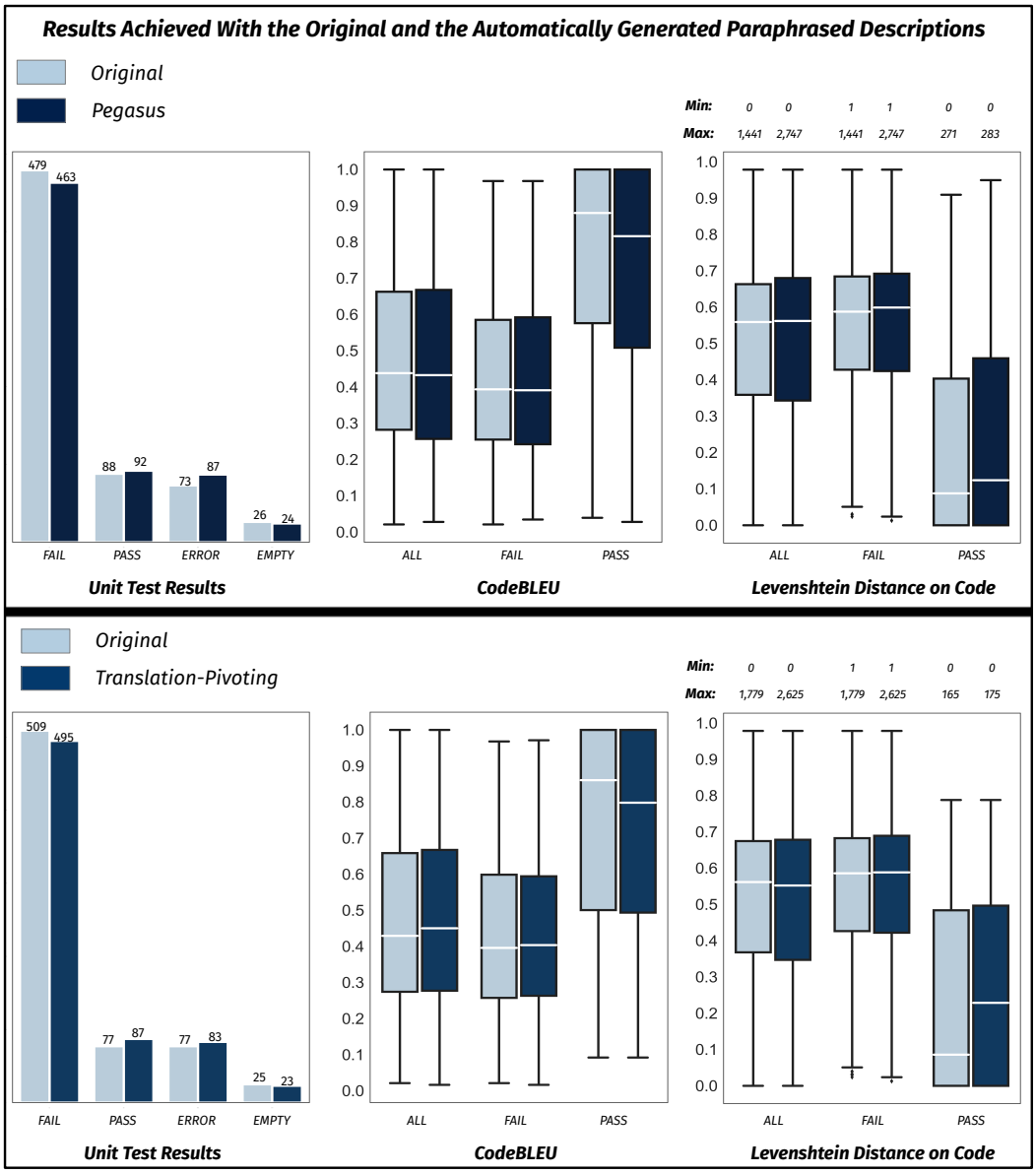


Figure 4.2. Results achieved by Copilot when considering the *Full context* code representation on $paraphrases_{PEGASUS}$ and $paraphrases_{TP}$.

Performance of Copilot when using the original and the paraphrased description as input. Fig. 4.4 summarizes the performance achieved by *Copilot* when using the *original* description (light blue) and the manually generated *paraphrased* description (dark blue) as input. Similarly, we report in Fig. 4.2 the performance obtained when considering the paraphrases generated with the two automated techniques, *i.e.,* PEGASUS and TP (top and bottom of Fig. 4.2, respectively). It is worth noticing that, in the latter, we only considered in the analysis the paraphrases manually considered as equivalent in RQ_0, *i.e.,* 666 for PEGASUS and 688 for TP.

A first interesting result is that, as it can be noticed from Fig. 4.4 and Fig. 4.2, the results obtained with the three methodologies are very similar. For this reason, to avoid repetitions, in the following, we will mainly focus on the results obtained with the manually generated paraphrases.

Also, as we will discuss, the quality of *Copilot*'s recommendations is very similar when using the *original* and the *paraphrased* descriptions.

Target Method

```
public void removeListener(IChemObjectListener col){
    if (chemObjectListeners == null) {
        return;
    }
    List<IChemObjectListener> listeners = lazyChemObjectListeners();
    if (listeners.contains(col)) {
        listeners.remove(col);
    }
}
```

Recommended method starting from the original description

PASS

```
public void removeListener(IChemObjectListener col){
    if (chemObjectListeners == null) {
        return;
    }
    lazyChemObjectListeners().remove(col);
}
```

CodeBLEU: 0.45

Figure 4.3. Example of recommended method that passes the unit tests but reports a low CodeBLEU score compared to the oracle (*i.e.,* target method).

In Fig. 4.4, the bar chart in the left side reports the number of methods recommended by *Copilot* (out of 892) that resulted in failing tests, passing tests, syntactic errors, and no (*i.e.,* empty) recommendation. Looking at such a chart, the first thing that leaps to the eyes is the high percentage of Java methods (∼73% for the *original* and ∼72% for the *paraphrased* description) for which *Copilot* was not able to synthesize a method passing the related unit tests.

Only ∼13% of instances (112 and 122 depending on the used description) resulted in test-passing methods. While such a result seems to indicate limited performance of *Copilot*, it must be considered the difficulty of the code generation tasks involved in our study. Indeed, we did not ask *Copilot* to generate simple methods possibly implementing quite popular routines (*e.g.,* a method to generate an MD5 hash from a string) but rather randomly selected

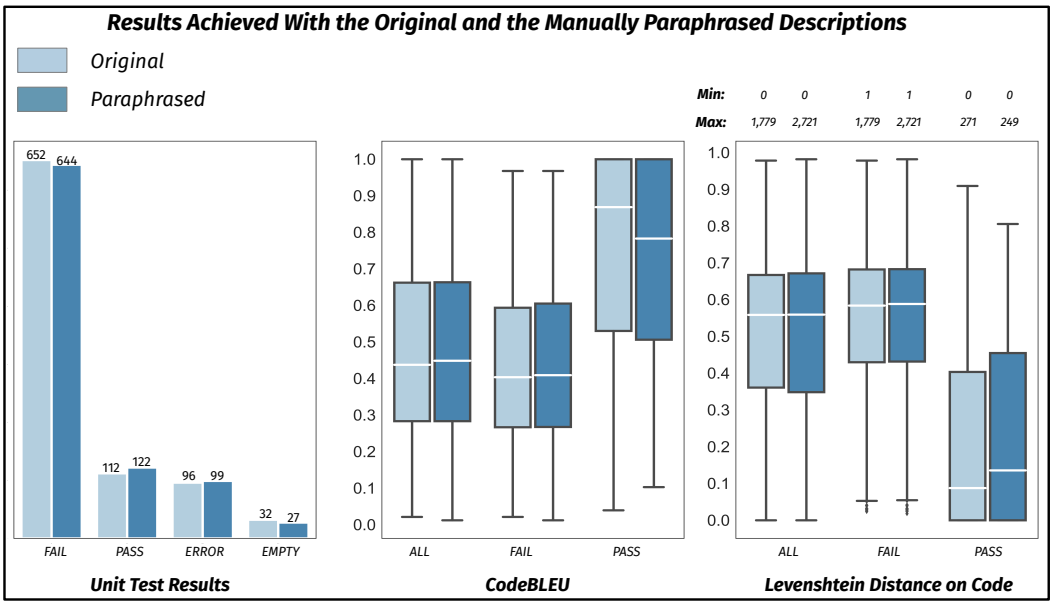


Figure 4.4. Results achieved by Copilot when considering the *Full context* code representation on $paraphrases_{manual}$.

methods that, as shown in Table 4.1, are composed, on average, by more than 150 tokens (median = 92) and have an average cyclomatic complexity of 5.3 (median = 3.0).

Thus, we consider the successful generation of more than 110 of these methods a quite impressive result for a code recommender. The remaining ∼15% of instances resulted either in a parsing error (∼100 methods) or in an empty recommendation (∼30 methods).

The box plot in the middle part of Fig. 4.4 depicts the results achieved in terms of CodeBLEU [RGL+20] computed between the recommended methods and the target one (*i.e.,* the one implemented by the original developers). Higher values indicate higher similarity between the compared methods. Instead, in the right box plot, we show the normalized Levenshtein distance, for which lower values indicate higher similarity.

For both metrics, we depict the distributions when considering all generated predictions, the ones failing tests, and the ones passing tests. As expected, higher (lower) values of CodeBLEU (Levenshtein distance) are associated with test-passing methods. Indeed, for the latter, the median CodeBLEU is ∼0.80 (Levenshtein = ∼0.10) as compared to the ∼0.40 (Levenshtein = ∼0.58) of test-failing methods. Despite such an expected finding, it is interesting to notice that 25% of test-passing methods have a rather low CodeBLEU <0.50.

Fig. 4.3 shows an example of recommended method having a CodeBLEU with the target method of 0.45 and passing the related tests. The recommended method, while substantially different from the target, captures the basic logic implemented in it. The target method first checks if the object `chemObjectListeners` is `null` and, if not, it proceeds removing from the `listeners` list the element matching the one provided as parameter (*i.e.,*`col`). The method synthesized by *Copilot* avoids the second `if` statement by directly performing the remove operation after the `null` check.

Note that there the two implementations are equivalent: The `remove` method of `java.util.List` preliminarily checks whether the passed element is contained in the list before removing it. While the check in the original method has no functional role, together with the introduction of the `listeners` variable, it might have been introduced to make the method more readable and self-explanatory.

Similarly, Fig. 4.5 shows an example of prediction passing the tests but that, accordingly to the Levenshtein distance, would require 165 token-level edits to match the target prediction (NTLev=63%). Differently from the previous example, it is clear that, in this case, the two methods do not have the same behavior since the recommended one also treats 3D points, while the original one only 2D points. In other words, the tests fail to capture the difference in the behavior. These examples provide two interesting observations. The first is that, metrics such as CodeBLEU and Levenshtein distance may result in substantially wrong assessments of the quality of a prediction. Indeed, while the discussed predictions have low CodeBLEU/high Levenshtein values and, thus, would be considered as unsuccessful predictions in most of the empirical evaluations, it is clear that they are valuable recommendations for a developer, even when not 100% correct (see Fig. 4.5). This poses questions on the usage of these metrics in the evaluation of code recommenders. Second, also the testing-based evaluation shows, as expected, some limitations as in the second example, in which the two methods do not implement the same behavior but both pass the tests.

As a final note, it is also interesting to observe as 25% of test-failing predictions exhibit high values ($>\sim$0.60) of CodeBLEU, indicating a high code similarity that, however, does not reflect in test-passing recommendations.

Impact of paraphrasing the input descriptions. Out of the 892 manually paraphrased descriptions, 408 (46%) result in different code recommendations as compared to the *original* description. This means that *Copilot* synthesizes different methods when it is provided as input with the *original* description and with the manually *paraphrased* description, which are supposed to summarize the same piece of code. Note that at this stage we are not focusing on the "quality" of the obtained predictions in any way. We are just observing that different input descriptions have indeed an impact on the recommended code. This implies that developers using different wordings to describe a needed method may end up with different recommendations. Such differences also result in the potential loss of correct recommendations. Indeed, out of the 112 test-passing predictions obtained with the *original* description and the 122 obtained with the manually *paraphrased* description, only 98 are in overlap, indicating that there are 38 correct recommendations only generated either by the *original* (14) or the *paraphrased* (24) description.

To have a deeper look into the 408 different predictions generated by *Copilot* with the *original* and the *paraphrased* description, the left part of Fig. 4.6 (light blue) shows the normalized token-level Levenshtein distance between (i) the *original* description and the *paraphrased* description (see the boxplot labeled with "Description"), and (ii) the method obtained using the *original* description and that recommended using the *paraphrased* description ("Code"). The "Description" boxplot depicts the percentage of words that must be changed to convert the *paraphrased* description into the *original* one. As it can be seen, while describing the same method, the *paraphrased* descriptions can be substantially different as

Target Method

```
public static void translateAllPositive(IAtomContainer atomCon){
    double minX = Double.MAX_VALUE;
    double minY = Double.MAX_VALUE;
    Iterator<IAtom> atoms = atomCon.atoms().iterator();
    while (atoms.hasNext()) {
        IAtom atom = (IAtom) atoms.next();
        if (atom.getPoint2d() != null) {
            if (atom.getPoint2d().x < minX) {
                minX = atom.getPoint2d().x;
            }
            if (atom.getPoint2d().y < minY) {
                minY = atom.getPoint2d().y;
            }
        }
    }
    logger.debug("Translating: minx=" + minX + ", minY=" + minY);
    translate2D(atomCon, minX * -1, minY * -1);
}
```

Recommended method starting from the original description

PASS

```
public static void translateAllPositive
     (IAtomContainer atomCon){
        double minX = Double.POSITIVE_INFINITY;
        double minY = Double.POSITIVE_INFINITY;
        double minZ = Double.POSITIVE_INFINITY;
        for (IAtom atom : atomCon.atoms()) {
            if (atom.getPoint2d() != null) {
                minX = Math.min(minX, atom.getPoint2d().x);
                minY = Math.min(minY, atom.getPoint2d().y);
            }
            if (atom.getPoint3d() != null) {
                minX = Math.min(minX, atom.getPoint3d().x);
                minY = Math.min(minY, atom.getPoint3d().y);
                minZ = Math.min(minZ, atom.getPoint3d().z);
            }
        }
        for (IAtom atom : atomCon.atoms()) {
            if (atom.getPoint2d() != null) {
                 atom.setPoint2d(new Point2d(
                                 atom.getPoint2d().x - minX,
                                 atom.getPoint2d().y - minY));
            }
            if (atom.getPoint3d() != null) {
                atom.setPoint3d(new Point3d(
                                atom.getPoint3d().x - minX,
                                atom.getPoint3d().y - minY,
                                atom.getPoint3d().z - minZ));
            }
        }
}
```

Levenshtein distance: 165

Figure 4.5. Example of recommended methods that pass the unit tests but would require 165 edit actions to match the target method.

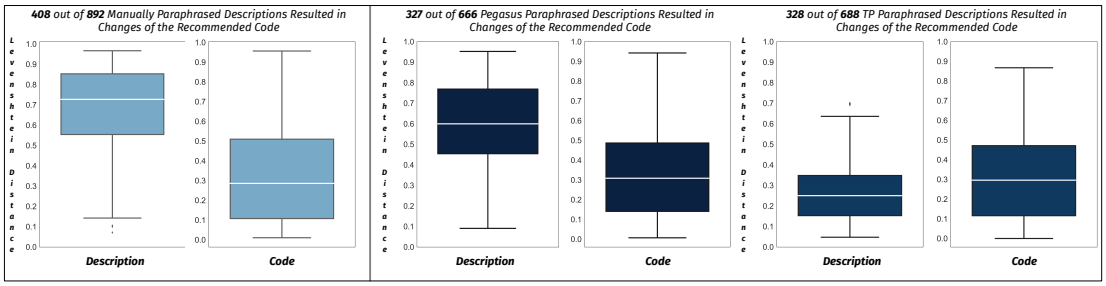


Figure 4.6. Levenshtein distance between the *original* description and (i) the manually *paraphrased* descriptions (left part) and (ii) the descriptions automatically paraphrased by PEGASUS (middle part) and Translate Pivoting (right). Similarly, we report the Levenshtein distance between the method recommended using the *original* description and the three paraphrases. The latter is only computed for recommendations in which the obtained output differs.

compared to the *original* ones, with 50% of them requiring changes to more than 70% of their words. Similarly, the different methods recommended in the 408 cases under analysis, can be substantially different, with a median of ∼30% of code tokes that must be changed to convert the recommendation obtained with the *original* description into the one obtained using the *paraphrased* description (see the "Code" boxplot).

These findings are confirmed for the automatically paraphrased descriptions (see the middle and the right part of Fig. 4.6 for the results achieved with the PEGASUS and TP paraphrases, respectively). As it can be seen, the main difference as compared to the results of the manually paraphrased description (left part of Fig. 4.6) is that TP changes a substantially lower number of words in the *original* description as compared to PEGASUS and to the manual paraphrasing. Such a finding is expected considering that TP just translates the *original* description back and forth from English to French, thus rarely adding new words to the sentence, something that is likely to happen using PEGASUS or by paraphrasing the sentence manually.

Answer to RQ$_1$. Different (but semantically equivalent) natural language descriptions of the same method are likely to result in different code recommendations generated by DL-based code generation models. Such differences can result in a loss of correct recommendations (∼28% of test-passing methods can only be obtained either with the *original* or the *paraphrased* descriptions). These findings suggest that testing the robustness of DL-based code recommenders may play an important role in ensuring their usability and in defining possible guidelines for the developers using them.

4.3 Threats to Validity

Threats to *construct validity* concern the relationship between the theory and what we observe. Concerning the performed measurements, we exploit the passing tests as a proxy for the correctness of the recommendations generated by *Copilot*. We acknowledge that

passing tests does not imply code correctness. However, this it can provide hints about the code behavior. To partially address this threat we focused our study on methods having high statement coverage (median = 100%). Also, we complemented this analysis with the CodeBLEU and the normalized token-level Levenshtein distance. As for the execution of our study, we automatically invoked *Copilot* rather than using it as actual developers would do: We automatically accepted the whole recommendations and did not simulate a scenario in which a developer selects only parts of the provided recommendations. In other words, while our automated script simulates a developer invoking *Copilot* for help, it cannot simulate the different usages a developer can make of the received code recommendation.

Threats to *internal validity* concern factors, internal to our study, that could affect our results. While in RQ_2 we had multiple authors inspecting the semantic equivalence of the paraphrasing generated by the automated tools, in RQ_1 we relied on a single author to paraphrase the *original* description. This introduces some form of subjectivity bias. However, the whole point of our paper is that, indeed, subjectivity plays a role in the natural language description of a function to generate and we are confident that the written descriptions were indeed semantically equivalent to the *original* one. Indeed, the authors involved in the manual paraphrasing have an average of seven years of experience in Java. Also related to internal validity is our choice of using the first sentence of the Doc Comments as the *original* natural language description. These sentences may be of low quality and not representative of how a developer would describe a method they want to automatically generate. This could substantially influence our findings, especially in terms of the effectiveness of *Copilot* (*i.e.,* its ability to generate test-passing methods). However, such a threat is at least mitigated by the fact that *Copilot* has also been invoked using the manually written descriptions, showing a similar effectiveness. A final threat regards the projects used for our study.

Those are open-source projects from GitHub, and it is likely that at least some of them have been used for training Copilot itself. In other words, the absolute actual effectiveness reported might not be reliable. However, the objective of our study is to understand the differences when different paraphrases are used rather than the absolute performance of Copilot, like previous studies did (*e.g.,* [NN22]).

Threats to *external validity* are related to the possibility to generalize our results. Our study has been run on 892 methods we carefully selected as explained in Section 4.1.1. Rather than going large-scale, we preferred to focus on methods having a high test coverage and a verbose first sentence in the Doc Comment. Larger investigations are needed to corroborate or contradict our findings. Similarly, we only focused on Java methods, given the effort required to implement the toolchain needed for our study, and in particular the script to automatically invoke *Copilot* and parse its output. Running the same experiment with other languages is part of our future agenda.

4.4 Conclusions

We investigated the extent to which DL-based code recommenders tend to synthesize different code components when starting from different but semantically equivalent natural language descriptions. We selected *GitHub Copilot* as the tool representative of the state-

of-the-art and asked it to generate 892 non-trivial Java methods starting from their natural language description. For each method in our dataset we asked *Copilot* to synthesize it using: (i) the *original* description, extracted as the first sentence in the Javadoc; and (ii) *paraphrased* descriptions. We did this both by manually modifying the *original* description and by using automated paraphrasing tools, after having assessed their reliability in this context.

We found that in ∼46% of cases semantically equivalent but different method descriptions result in different code recommendations. We observed that some correct recommendations can only be obtained using one of the semantically equivalent descriptions as input.

Our results highlight the importance of providing a proper code description when asking DL-based recommenders to synthesize code. In the new era of AI-supported programming, developers must learn how to properly describe the code components they are looking for to maximize the effectiveness of the AI support.

Part III

Automated Log Generation

Inspecting log messages is a popular practice that helps developers in several software maintenance activities such as testing [CSX+18, CSH+19], debugging [SSKM92], diagnosis [ZPX+19, YZP+12], and monitoring [HvH20, HZW+21]. Developers insert log statements to expose and register information about the internal behavior of a software artifact in a human-comprehensible fashion [OGX12]. The data generated is used for runtime and post-mortem analyses. For example, when debugging, log statements can support root cause analysis [LRW+17, GJL+16], while once the software is deployed logs can be used for performance monitoring [YBdPS+18] or anomaly detection [MLZ+19, ZXL+19, DLZS17].

Researchers have proposed techniques to support developers in deciding what parts of the system to log [YMX+10], the log level for logging statements [LSA+20, YPZ12, OGX12, LSH17], and the structure of log messages [Li20]. While achieving great performance, these techniques only partially support developers in logging practices. Indeed, none of them can generate complete log statements providing to the developer (i) the location where to inject it, (ii) the correct log level to use, and (iii) the actual log statement also featuring the needed natural language log message.

To overcome these limitations, we presented LANCE (**L**og st**A**teme**N**t re**C**omm**E**nder) [MPB22], an approach exploiting the Text-To-Text-Transfer-Transformer (T5) model [RSR+20] to automatically generate and inject complete logging statements in *Java* code. We started by pre-training our model on a set of 6,832,859 *Java* methods. Once pre-trained, the model has been fine-tuned to generate complete log statements. In particular, given a *Java* method as input to LANCE, we ask it to inject a complete log statement where needed. This means that LANCE must generate a complete log statement and inject it in the proper location. In our evaluation, we asked LANCE to automatically generate 12,020 log statements and compared them to the ones manually written by developers. We found that LANCE is able to (i) correctly predict the appropriate location of a log statement in 65.9% of cases; (ii) select a proper log level for the statement in 66.2% of cases; and (iii) generate a completely correct logging statement, including a meaningful natural language message in 15.2% of cases. The results of this work have been presented in the following publications:

> **Using Deep Learning to Generate Complete Log Statements**
> Antonio Mastropaolo, Luca Pascarella, Gabriele Bavota. In *Proceedings of the 44th IEEE/ACM International Conference on Software Engineering (ICSE 2022)*, pp. 2279-2290

While LANCE represented a major step ahead in log automation as compared to state-of-the-art techniques, it suffers of two major limitations. First, it assumes that only one log statement is needed in a *Java* method provided as input. This is due to the training procedure we employed that asks the model to always generate a single log statement. Second, given a *Java* method, LANCE cannot assess whether log statements are needed at all. Indeed, in some cases, enough log statements may be already present in the method or, maybe, the method does not feature statements that would benefit from logging. For this reason, we presented an extension of LANCE, named LEONID, that combines DL and Information

Retrieval (IR) techniques to overcome these limitations. LEONID can not only discriminate between methods that need log statements and those that do not but also supports the injection of multiple log statements where needed. LEONID achieves a success rate of 27.27% in correctly injecting a single log statement into *Java* methods (as compared to the 15.2% of LANCE), while it succeeds in 17% of the cases when dealing with methods in need of multiple log statements. LEONID has been presented in the following publication:

> **Log statements generation via deep learning: Widening the support provided to developers**
>
> Antonio Mastropaolo, Valentina Ferrari, Luca Pascarella, Gabriele Bavota. In *Proceedings of Journal of Systems and Software (JSS 2023)*, Volume 210(4)

In the following chapters we first discuss the literature related to the automation of logging activities (Chapter 5) and then present our latest approach on log automation, namely LEONID (Chapter 6). Chapter 6 also features a direct comparison between LEONID and LANCE [MPB22], our first proposal for log automation.

5

Background and Related Work

We discuss the related literature revolving around approaches designed to support and automate logging activities. We omit empirical studies on logging practices [YPZ12, FZH+14, CJ17, ZCSC19, YPZ12, ZYD+19, ZHCHL20, LSA+20], since not directly related to our research.

Log message enhancement. A first family of techniques aim at recommending how to improve the log message reported in a given log statement. Yuan *et al.* [YZP+12] proposed LOGENHANCER as a prototype to automatically recommend relevant variable values for each log statement, refactoring its message to include such values. Their evaluation on eight systems demonstrates that LOGENHANCER can reduce the set of potential root failure causes when inspecting log messages. Liu *et al.* [LXL+19] tackled the same problem using, however, a customized deep learning network. Their evaluation showed that the mean average precision of their approach is over 84%.

Ding *et al.* proposed LOGENTEXT [DLS22], a NMT (Neural Machine Translation) approach for improving the quality of log messages: By taking the code preceding a given log statement, LOGENTEXT can translate it into a short textual description that can be used for logging. In a subsequent study, Ding *et al.* [DTC+23] expand on LogGenText by developing LoGenText-Plus. This improved technique breaks down the process of generating logging text into two phases. Initially, LoGenText-Plus leverages an NMT model to create the syntactic template for the intended logging text. Then, it inputs both the source code and this generated template into another NMT model dedicated to the synthesis of logging text.

Log placement. Other researchers targeted the suggestion of the best code location for log statements [JLL+18, LCSH18, Li20]. For example, Zhu *et al.* [ZHF+15] presented LOGADVISOR, an approach to recommend where to add log statements. The evaluation of LOGADVISOR on two Microsoft systems and two open-source projects reported an accuracy of 60% when applied on pieces of code without log statements. Yao *et al.* [YBdPS+18] tackled the same problem in the specific context of monitoring the CPU usage of web-based systems, showing that their approach helps developers when logging.

Li *et al.* [LCS20] proposed a deep learning framework to recommend logging locations at the code block level. They report a 80% accuracy in suggesting logging locations using

within-project training, with slightly worse results (67%) in a cross-project setting. Cândido *et al.* [CHAvD21] investigated the effectiveness of log placement techniques in an industrial context. Their findings (*e.g.,* 79% of accuracy) show that models trained on open source code can be effectively used in industry.

Log level recommendation. A third family of techniques focus on recommending the proper log level (*e.g.,* error, warning, info) for a given log statement [YPZ12, OGX12]. Mizouchi *et al.* [MSII19] proposed PADLA as an extension for Apache Log4j framework to automatically change the log level for better record of runtime information in case of anomalies. The DEEPLV approach proposed by Li *et al.* [LLCS21] uses instead a deep learning model to recommend the level of existing log statements in methods. DEEPLV aggregates syntactic and semantic information of the source code and showed its superiority with respect to the state-of-the-art.

Generating complete log statements. Finally, techniques have been proposed to provide support for all logging tasks (*i.e.,* log placement, log message creation, and log level recommendation), allowing the automated injection of complete log statements. Xu *et al.* [XCZ+24] utilize LLMs and in-context learning to facilitate all three logging tasks in a seamless, end-to-end manner. Specifically, they introduce UNILOG, an innovative logging framework that applies the in-context learning approach to guide LLMs in task-specific knowledge application. The findings from their experiments highlight that carefully crafted prompts for the complete automation of logging tasks can effectively set new benchmarks in the field.

Li *et al.* [LHZ+24] introduce SCLOGGER, an approach using contextual information to generate comprehensive logging statements. SCLogger makes use of static domain knowledge through context-aware prompts and employs recent techniques such as chain-of-thoughts [WWS+22]. The outcomes of their evaluation demonstrate that SCLogger surpasses the state-of-the-art benchmarks [MPB22] in producing complete log statements.

Contribution in the area. When looking at solutions generating complete log statements, we presented LANCE [MPB22] and its extension LEONID [MFPB24], detailed in Chapter 6. LANCE was the pioneering technique in the literature to support the generation of complete log statements and represented the baseline for comparison in subsequent proposals [LHZ+24].

6

Log Statement Generation via Deep Learning

Logging poses several challenges to software developers. First, they need to decide *what to log*, by finding the right amount of log statements needed in the application without, however, flood it with useless log statements. Second, developers must *log at the proper level*, namely select the proper log level for each entry (*e.g.,* info, warning, error). Third, log statements must be accompanied by *meaningful and informative* log messages that can be easily understood.

In our first attempt to automate logging activities [MPB22], we presented LANCE, an approach built on top of a T5 [RSR+20] deep learning model trained to generate and inject a complete log statement in a *Java* method provided as input. T5 has been pre-trained on a set of ∼6.8M *Java* methods using the classic "masked language modeling" objective [RSR+20]. In the case of LANCE, this means that during pre-training the model is provided as input a *Java* method with 15% of its tokens masked and it is expected to predict the masked tokens. Such a pre-training task provides T5 with knowledge about the language of interest (*i.e., Java*). Once pre-trained, the model has been fine-tuned for the specific task of interest. In this case, we selected ∼62k *Java* methods and removed from them exactly one log statement asking the model to generate and inject it, thus deciding *where* to log (*i.e.,* in which part of the method), which *log level* to use, and *what* to log (*i.e.,* generate a meaningful log message in natural language). The training procedure used for LANCE resulted in two strong limitations: First, LANCE assumes that only one log statement is needed in a *Java* method provided as input, since during training we asked the model to always generate a single log statement. Second, given a *Java* method, LANCE cannot assess whether log statements are needed at all. In this chapter, we present LEONID, an approach aimed at partially addressing these limitations.

We start replicating LANCE by training and testing it on a dataset 3.6 times larger than the one we used originally [MPB22] (230k training instances *vs* 63k). Besides being larger, the new dataset features a more variegate set of log statements. Then, we present LEONID as an extension of LANCE able to (i) discriminate between methods *needing* and *not needing* the injection of new log statements; and (ii) in case a need for log statements is identified, LEONID, differently from LANCE, can decide the proper number of log statements to inject (which can be higher than one) and properly place them in the correct position. We found

Table 6.1. State-of-the-art approaches supporting developers in logging activities

Ref.	Venue	Name	Log			Log injection		Need for log statements
			Level	Position	Message	Single	Multiple	
Zhu *et al.* [ZHF+15]	ICSE 2015	LogAdvisor	✗	✓	✗	✓	✗	✗
Yao *et al.* [YBdPS+18]	ICPE 2018	Log4Perf	✗	✓	✗	✓	✗	✓
Mizouchi *et al.* [MSII19]	ICPC 2019	PADLA	✓	✗	✗	✓	✓	✗
Li *et al.* [LCS20]	ASE 2020		✗	✓	✗	✓	✗	✗
Li *et al.* [LLCS21]	ICSE 2021	DeepLV	✓	✓	✗	✓	✗	✗
Ding *et al.* [DLS22]	SANER 2022	*LoGenText*	✗	✗	✓	✓	✗	✗
Mastropaolo *et al.* [MPB22]	ICSE 2022	LANCE	✓	✓	✓	✓	✗	✗
Our work	-	**LEONID**	✓	✓	✓	✓	✓	✓

that LEONID can correctly predict the need for log statements with an accuracy higher than 90%. Also, when log statements are needed, LEONID can generate and inject in the right position multiple complete log statements in ∼17% of cases.

Finally, in LEONID we attempted to improve the performance achieved in the generation of meaningful log messages by exploiting a combination of DL and Information Retrieval (IR). Indeed, based on the results we achieved with LANCE, the generation of log messages really looked like the Achilles' heel of DL-based log generation. Results show that by increasing the size of the training dataset, the ability of LANCE in predicting meaningful log messages substantially improves (+100% as compared to what we reported in [MPB22]). Instead, the combination of DL and IR we propose in LEONID only marginally improves the results for this specific task (+5% relative improvement).

Table 6.1 shows how LEONID widened the support provided to developers in the automation of logging activities as compared to the existing state-of-the-art techniques present at the time LEONID has been proposed.

LEONID is publicly available as a Visual Studio Code plugin. [1]

6.1 LEONID

6.1.1 Datasets Needed for Training, Validation, and Testing

We start by describing the dataset used for pre-training T5 (Section 6.1.2). Then, we detail the several fine-tuning datasets we built (featuring training, validation, and test set). The first, aimed at replicating LANCE [MPB22], teaches T5 how to inject a single log statement in a *Java* method (Section 6.1.3). The second fine-tuning dataset also focuses on the problem of injecting a single log statement, but this time exploits IR to provide T5 with concrete examples of log messages that might be relevant for the prediction at hand (Section 6.1.4). This allows to compare LANCE with LEONID in the task of single log statement injection. The third fine-tuning dataset trains LEONID for the task of multi-log statements prediction, *i.e.,* injecting from 1 to *n* log statements in a given method (Section 6.1.5). Finally, we describe the fine-tuning dataset to train a T5 able to discriminate between methods *needing* and *not needing* log statements (Section 6.1.6). The datasets are summarized in Tables 6.2 and 6.3 and available in [Mas23].

[1] LEONID can be found at: `https://marketplace.visualstudio.com/items?itemName=AndreaMicheleZucchi.loginjector`

All datasets have been built starting from the same set of GitHub repositories that we selected using the GHS (GitHub Search) tool by Dabić *et al.* [DAB21]. GHS allows to query GitHub for projects meeting specific criteria. We used the same selection criteria exploited in our former work on LANCE [MPB22], selecting all public non-forked *Java* projects having at least 500 commits, 10 contributors, and 10 stars. These selection criteria aim at excluding personal/toy projects and reduce the chance of collecting duplicated code (non-forked repositories). We cloned the latest snapshot of the 6,352 projects returned by GHS. We scanned all cloned repositories to assess whether they featured a `POM` (Project Object Model) or a `build.gradle` file. Both these files allow to declare external dependencies towards libraries, the former using Maven, the latter Gradle. Such a check was performed since, as a subsequent step, we verify whether projects had a dependency towards Apache Log4j [Lognd] (*i.e.,* a well-known *Java* logging library) or SLF4J (Simple Logging Facade for *Java*) [QOSnd] (*i.e.,* an abstraction for *Java* logging frameworks similar to Log4j). Indeed, to train a T5 for the task of injecting complete log statement(s) in *Java* methods, we need examples of methods featuring log statements. The usage of popular logging *Java* libraries was thus a prerequisite for the project's selection.

We found 3,865 projects having either a `POM` or a `build.gradle` file and 2,978 of them featured a dependency towards at least one logging library. The overall projects' selection is very similar to the one we performed in [MPB22], with the main differences being the additional mining of projects: (i) using Gradle as build system (in [MPB22] only Maven was considered); and (ii) having a dependency towards SLF4J (in [MPB22] only Log4j was considered). These choices help in increasing the size and variety of both the training and the testing datasets, making the prediction more challenging.

We used srcML [src] to extracted all *Java* methods in the selected projects. Then, we identified the log statements within each method (if any) and removed all methods featuring log statements exploiting custom log levels (*i.e.,* log levels that do not belong to any of the two libraries we consider, but that have been defined within a specific project). The valid log levels we considered are: `FATAL`, `ERROR`, `WARN`, `DEBUG`, `INFO`, and `TRACE`. At this point we were left with two sets of methods: those not having any log statement and those having at least one log statement using one of the "valid" log levels.

We run *javalang* [Thund] on these methods to tokenize them and excluded all those having $\#tokens < 10$ or $\#tokens \geq 512$. The upper-bound filtering has been done in previous works [MAPB21, TPT+21, CCP+21, TWB+19b, TWB+19a] to limit the computational expenses of training DL-based models. The lower-bound of 10 tokens aims at removing empty methods. We also removed all methods containing non-ASCII characters in an attempt to exclude at least some of the methods featuring log messages not written in English. Finally, to avoid any possible overlap between the training, evaluation, and test datasets we are going to create from the collected set of methods, we removed all exact duplicates, obtaining the final set of 12,916,063 *Java* methods, of which 244,588 contain at least one log statement.

Table 6.2. Number of methods in the datasets used in our study

Dataset	train		eval	test
	w/ log	w/o log	w/ log	w/ log
Pre-training	-	12,671,475	-	-
Fine-tuning: Single Log Generation	229,703	-	28,763	28,698
Fine-tuning: Single Log Generation with IR	229,703	-	28,763	28,698
Fine-tuning: Multi-log Injection with IR	192,773	-	24,092	24,088

6.1.2 Pre-Training Dataset

Since the goal of pre-training is to provide T5 with general knowledge about the language of interest (*i.e., Java*), we used for pre-training all methods not featuring a log statement (the latter will be used for the fine-tuning datasets). We adopted a classic *masked language model* task, which consists in randomly masking 15% of the tokens composing a training instance (*i.e.,* a *Java* method) asking the model to predict them.

Fig. 6.1 depicts the masking procedure of instances used to pre-train the model.

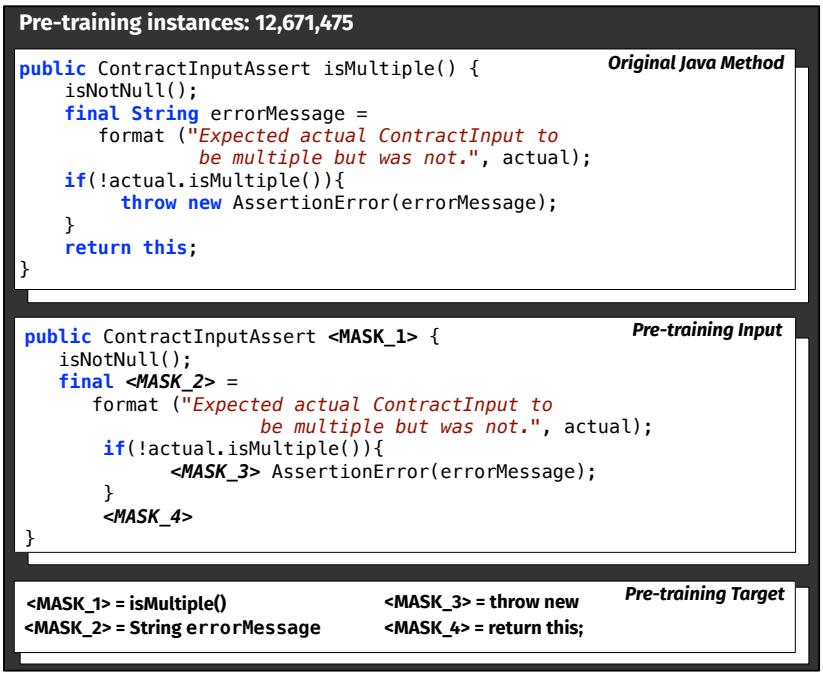


Figure 6.1. Example of Pre-training instance

6.1.3 Fine-tuning Dataset: Single Log Generation

We build a fine-tuning dataset aimed at replicating what we did in the training of LANCE [MPB22]. We process each method M having $n \geq 1$ log statements by removing from it one log statement (*i.e.,* leaving it with $n-1$ log statements). This allows to create a training pair

Table 6.3. Number of methods in the datasets used to predict the need for log statements

Dataset	train		eval		test	
	Need	**No need**	**Need**	**No need**	**Need**	**No need**
Fine-tuning: Need4Log (50-50)	98,848	92,126	12,257	11,468	11,627	11,627
Fine-tuning: Need4Log (75-25)	98,848	92,126	12,257	11,468	12,159	4,053
Fine-tuning: Need4Log (25-75)	98,848	92,126	12,257	11,468	3,875	11,627
Fine-tuning: Need4Log (2-98)	98,848	92,126	12,257	11,468	238	11,627

$\langle M_s, M_t \rangle$ with M_s representing the input provided to the model (*i.e., M* with one removed log statement) and M_t being the expected output (*i.e., M* in its original form, with all its log statements). This is the dataset used to train LANCE [MPB22] and it allows to train a model able, given a *Java* method as input, to inject in it one new log statement. For methods having $n > 1$ (*i.e.,* more than one log statement), we created n pairs $\langle M_s, M_t \rangle$, each of them having one of the n log statements removed (*i.e.,* different M_s). To ensure that after the log statement removal our instances still featured valid *Java* methods, we parsed each M_s using JavaParser [Javnd] and removed all pairs including an invalid M_s.

We split the remaining pairs into training (80%), validation (10%) and test (10%) set as reported in Table 6.2. Training and testing a T5 model on this dataset basically means performing a differentiated replication of LANCE on a 3.6× larger and more variegate (multiple logging libraries) dataset.

6.1.4 Fine-tuning Dataset: Single Log Generation with IR

In LEONID, we combine DL and IR with the goal of boosting performance especially in the generation of meaningful log messages. The main idea is to augment the input provided to the model (*i.e.,* M_s) with log messages belonging to methods similar to M_s which are featured in the training set. For each of the 244,588 $\langle M_s, M_t \rangle$ pairs in the fine-tuning dataset described in Section 6.1.3 (this includes training, validation, and test), we identify the k most similar pairs in the training set. The similarity between two pairs is based on the similarity of their M_s (*i.e.,* the method in which the log statement must be created) and it is computed using the Jaccard similarity [Han04] index, based on the percentage of code tokens shared across the two methods. We then use these k similar methods to extract from them examples of log messages used in coding contexts which are similar to the M_s at hand.

Two clarifications are needed. First, independently if a given pair is in the training, validation, or test set, we extract its k most similar pairs only from the training set. This is needed since, while predicting the log statement to inject, the training set must be the only knowledge available to the model (*i.e.,* the test set must be composed of previously unseen instances). Second, when computing the Jaccard similarity, we remove from the compared methods all log statements, since we want to identify similar "coding contexts" that may require similar log statements. We created three different fine-tuning datasets using different values of $k = \{1, 3, 5\}$ (thus, a lower/higher number of exemplar log messages provided to the model).

Fig. 6.2 shows an example of training instance for this fine-tuning dataset. The method

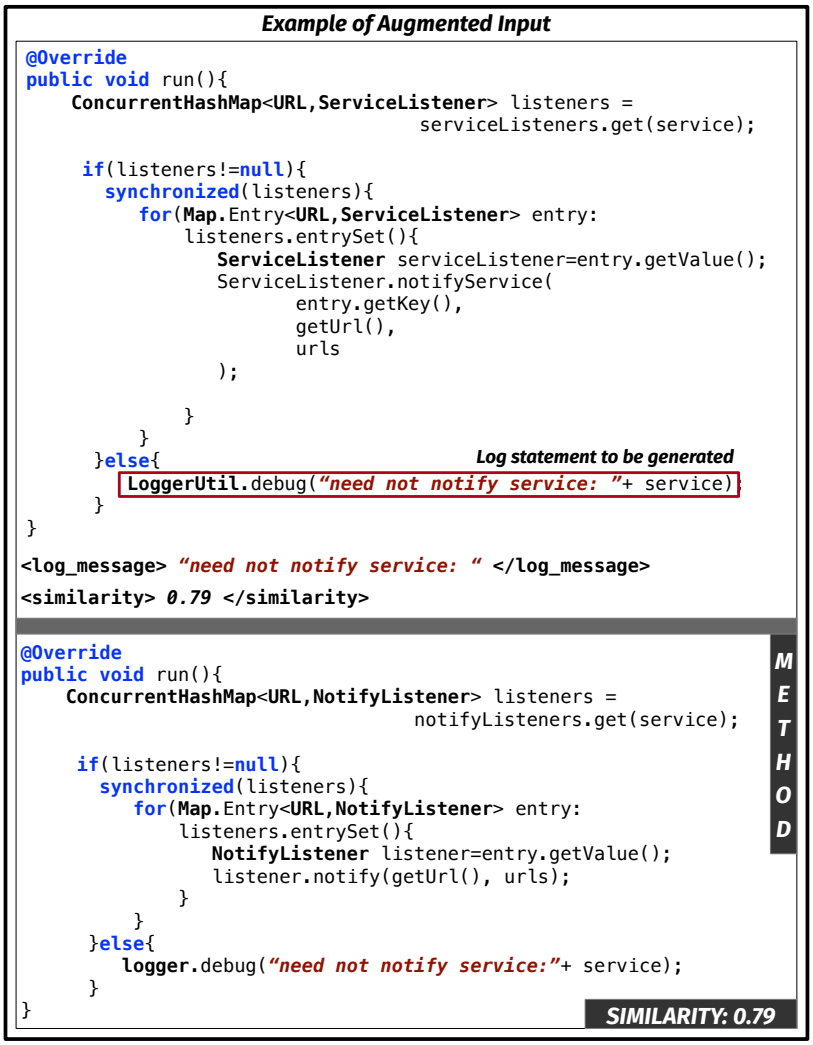


Figure 6.2. Example of instance in the "Single Log Generation with IR" dataset

on top represents the M_s *Java* method in which a log statement must be injected (*i.e.,* the one highlighted in red). The method is enriched with the exemplar log messages that have been found in the $k = 1$ most similar method shown in the bottom. Besides the log messages, we also provide T5 with the Jaccard similarity between the M_s at hand (top of the figure in this case) and the method of the training set from which the exemplar log message(s) has been extracted. This is meant to provide T5 with an additional hint in terms of which exemplar message comes from the most similar coding context (when more messages are retrieved).

Note that the instances in this dataset are exactly the same of the one previously described to replicate LANCE (see Table 6.2). This allows a direct comparison in terms of performance which will provide information about the gain, if any, provided by the IR integration.

6.1.5 Fine-tuning Dataset: Multi-log Injection with IR

One limitation of LANCE [MPB22] we aim at addressing in this extension is the assumption that a *Java* method provided as input always requires one new log statement to be injected.

Also for this dataset, LEONID exploits a combination of DL and IR, thus we follow a process similar to the one described in Section 6.1.4, with the main difference being the number of log statements we ask the model to generate. Given a method M featuring n log statements, we randomly select y log statements to remove from it, with $1 \leq y \leq n$. This means that we create pairs $\langle M_s, M_t \rangle$ in which M_s lacks a "random" number of log statements that must be generated by the model to obtain the target method M_t. This makes the prediction task substantially more challenging as compared to the single-log injection scenario experimented in LANCE. Also in this case we parsed each M_s using JavaParser [Javnd] and removed all pairs including an invalid M_s. The remaining part of the process (*i.e.,* identifying the k most similar pairs to inject examples of log messages) is the same described in Section 6.1.4. Table 6.2 shows the distribution of instances among the training, evaluation, and test set for this dataset as well.

6.1.6 Fine-tuning Dataset: Deciding Whether Log Statements are Needed

While the dataset described in Section 6.1.5 allows to build a model able to inject multiple log statements in a given *Java* method, such a model still assumes that at least one log statement must be injected in the input method. Thus, LEONID also includes a T5 model trained as a binary classifier in charge of deciding whether a method provided as input requires the addition of log statements or not. In case of affirmative answer, the method can then be passed to the previously trained model which will decide how many and which log statements to inject. To train such a classifier we again start from the original set of 244,588 *Java* methods having at least one log statement. Then, similarly to what done in Section 6.1.5, given a method M featuring n log statements, we randomly select y log statements to remove from it with, however, $0 \leq y \leq n$. Thus, differently from the training dataset used for multi-log injection, we have instances from which we did not remove any log statement ($y = 0$). Then, we create a pair $\langle M_s, B \rangle$ in which M_s is the original method M **possibly** lacking a random number of log statements, while B is a boolean variable that could be equal *true* (*i.e.,* M_s needs the addition of log statements, since $y \geq 1$) or *false* (*i.e.,* no log statements are needed in M_s, since $y = 0$). Non-parsable methods resulting after the removal of the log statements have then been removed, as well as duplicates resulting from different methods that, after the removal of log statements, become equal (*i.e.,* their only differences were the removed log statements). This process resulted in a dataset featuring 190,974 training instances (98,848 needing at least a log statement and 92,126 not needing it), accompanied by the evaluation and test sets summarized in Table 6.3.

As it can be seen, four different versions of the test set have been created, to experiment LEONID in different scenarios. Let us explain such a choice. The test set should be representative of the real distribution of methods *needing* and *not needing* log statements. However, such a distribution cannot be computed in a reliable way. Indeed, one possibility we considered to build our dataset was to just consider all methods with and without log statements

as training instances (as opposed to work only with methods having at least a log statement as we do). In a nutshell, the process would have been: (i) remove a random number of log statements from the methods with at least one log statement to create instances *needing* logs; and (ii) assume that all methods without log statements do not require logging. However, assuming that all methods in a project not having log statements do not require logging is a very strong assumption. It is indeed possible that the project's developers just did not consider yet the usage of logs in a specific method or that, in a given project, logging is not yet a practice at all (thus all methods do not use log statements). This makes difficult a reliable computation of the number of methods *needing* and *not needing* logging. Also, such a problem justifies our decision to create instances of methods *needing*/*not needing* a log statement starting from all methods having at least one log statement and using the process described above (*i.e.,* removing a random number of statements to create instances in need of logging, and not removing any log statement to create instances not needing logging). At least, we are sure that these are methods for which developers considered logging (since they have at least one log statement) and, thus, can be seen as a sort of "oracle".

The four test sets in Table 6.3 simulate four different distributions of methods *needing*/*not needing* log statements: balanced (50% per category), unbalanced towards *needing* (75%-25%), unbalanced towards *not needing* (25%-75%), and strongly unbalanced towards *not needing* (2%-98%). The latter is a distribution we computed based on all 12M+ methods we mined, in which 98% of methods do not have log statements, while 2% have it. As said, this distribution is not completely reliable but, at least, gives an idea of what we found in the mined projects.

6.1.7 Training and Hyperparameter Tuning

All training we performed have been run using a Google Colab's 2x2, 8 cores TPU topology with a batch size of 128. Since we use software-specific corpora for pre-training and fine-tuning, we trained a tokenizer (*i.e.,* a SentencePiece model [Kud18]) on 1M *Java* methods randomly extracted from the pre-training dataset and 712,634 English sentences from the C4 dataset [RSR+20]. We included English sentences since, once fine-tuned, the models may be required to synthesize complex (natural language) log messages. We set the size of the vocabulary to 32k word-pieces.

6.1.7.1 Pre-training

We pre-trained T5 for 500k steps on the pre-training dataset composed by 12,671,475 *Java* methods (Table 6.2). Given the size of our dataset and the batch size, 500k steps correspond to ∼5 epochs. The maximum size of the input/output was set to 512 tokens.

6.1.7.2 Hyperparameter Tuning

Once pre-trained the model, we finetune the hyperparameters of the model following the same procedure we employed when developing LANCE. Such a procedure has been executed for each of the fine-tuning datasets previously described. In particular, we assessed the

Table 6.4. T5 hyperparameter tuning results (in bold the best learning rate)

Experiment	C-LR	ST-LR	ISQ-LR	PD-LR
Fine-tuning: Single Log Generation with IR ($k = 1$)	24.63%	25.92%	**26.55%**	26.36%
Fine-tuning: Single Log Generation with IR ($k = 3$)	26.25%	26.04%	**26.68%**	26.33%
Fine-tuning: Single Log Generation with IR ($k = 5$)	26.24%	25.69%	**26.78%**	26.33%
Fine-tuning: Multi-log Generation with IR ($k = 1$)	22.62%	22.19%	**22.79%**	22.76%
Fine-tuning: Multi-log Generation with IR ($k = 3$)	22.64%	22.28%	**23.05%**	22.59%
Fine-tuning: Multi-log Generation with IR ($k = 5$)	22.71%	22.14%	**22.78%**	22.51%
Fine-tuning: Need4Log	96.58%	96.56%	96.59%	**96.62%**

performance of T5 when using four different learning rate scheduler: (i) *Constant Learning Rate* (C-LR): the learning rate is fixed during the whole training; (ii) *Inverse Square Root Learning Rate* (ISR-LR): the learning rate decays as the inverse square root of the training step; (iii) *Slanted Triangular Learning Rate [HR18]* (ST-LR): the learning rate first linearly increases and then linearly decays to the starting learning rate; and (iv) *Polynomial Decay Learning Rate* (PD-LR): the learning rate decays polynomially from an initial value to an ending value in the given decay steps. The exact configuration of all the parameters used for each scheduling strategy is reported in Table 6.5.

Learning Rate Type	Parameters
Constant	$LR = 0.001$
Inverse Square Root	$LR_{starting} = 0.01$
	$Warmup = 10{,}000$
Slanted Triangular	$LR_{starting} = 0.001$
	$LR_{max} = 0.01$
	$Ratio = 32$
	$Cut = 0.1$
Polynomial Decay	$LR_{starting} = 0.01$
	$LR_{end} = 0.001$
	$Power = 0.5$

Table 6.5. Configurations for the experimented learning rates

Each model has been run for 100k training steps on the fine-tuning dataset. Then, its performance has been assessed on the evaluation set in terms of correct predictions (*i.e.,* cases in which the generated output is equal to the target one).

For the generative models injecting log statements this means that they outputted the *Java* method featuring all correct log statements in the expected positions. For the classifier, it means that it correctly predicted the need for log statements in a given method. The results achieved with each learning rate are reported in Table 6.4. Our hyperparameter tuning required training and evaluating 28 models: For each of the 7 fine-tuning datasets in Table 6.4 we experimented 4 different learning rates. Given the achieved results, we will use the ISQ-LR for the generative models, and the PD-LR for the classifier when fine-tuning the models. Concerning the "replication of LANCE" (*i.e.,* fine-tuning T5 on the dataset *Fine-*

tuning: Single Log Generation in Table 6.2), we did not perform any hyperparameter tuning, but relied on the best configuration reported in the original paper [MPB22], thus using the PD-LR.

6.1.7.3 Fine-tuning

Once identified the best learning rates to use, we fine-tuned the final models using early stopping, with checkpoints saved every 10k steps, a delta of 0.01, and a patience of 5. This means training the model on the fine-tuning dataset and evaluating its performance (again in terms of correct predictions) on the evaluation set every 10k. The training process stops if a gain lower than delta (0.01) is observed at each 50k steps interval. This means that after 60k steps, the performance of the model is compared against that of the 10k checkpoint and, if the gain in performance is lower than 0.01, the training stops and the best-performing checkpoint up to that training step is selected. This process has been used for all models, including the one replicating LANCE. Our replication package [Mas23] reports the convergence of all models (*i.e.,* the steps after which the early stopping criterion was met).

6.1.8 Generating Predictions

Once the T5 models have been pre-trained and fine-tuned, they can be used to generate predictions for the targeted tasks. We generate predictions using a greedy decoding strategy, meaning that the generated prediction is the result of selecting at each decoding step the token with the highest probability of appearing in a specific position. Thus, a single prediction (*i.e.,* the one maximizing the likelihood of among all the produced tokens) is generated for an input sequence, as compared to strategies such as beam-search [FAO17] that generate multiple predictions.

6.2 Study Design

The *goal* of our study is to evaluate the performance of LEONID in supporting logging activities in *Java* methods. We focus on three scenarios: single log injection, in which we compare with our previous approach LANCE [MPB22]; multi-log injection; and deciding weather log statements are needed or not in a given *Java* method. The context is represented by the test datasets reported in Table 6.2 (single and multi-log injection) and Table 6.3 (deciding whether logging is needed).

We aim at answering the following research questions:

RQ_1: *To what extent is LEONID able to correctly inject a single complete logging statement in* Java *methods?* RQ_1 mirrors the study we performed when presenting LANCE. We experiment LEONID in the same scenario presented in [MPB22]: The injection of a single log statement in a given *Java* method. We compare the performance of LEONID with that of LANCE when training and testing them on the same dataset.

RQ_2: *To what extent is LEONID able to correctly inject multiple log statements when needed?* RQ_2 tests LEONID in the more challenging scenario of injecting from 1 to *n* log statements in a *Java* method, as needed.

RQ_3: *To what extent is LEONID able to properly decide when to inject log statements?* RQ_3 analyzes the accuracy of LEONID in predicting whether or not log statements are needed in a given *Java* method. Additionally, we assess LEONID as a whole using it to both predict the need for log statements and, subsequently, generate and inject them (if needed).

6.2.1 Data Collection and Analysis

To answer RQ_1 we run both LEONID and LANCE against the test set described in Table 6.2 for the single log generation task. The only difference is that LANCE has been trained on the dataset not featuring the exemplar log messages added through IR (row *Fine-tuning: Single Log Generation* in Table 6.2), while LEONID exploits this information (row *Fine-tuning: Single Log Generation with IR* in Table 6.2). However, the training and test instances are exactly the same, allowing for a direct comparison. We assess the performance of the two techniques using the same evaluation schema employed in [MPB22]. In particular, we contrast the predictions generated by the two models against the expected output (*i.e.,* the *Java* method provided as input with the addition of the correct log statement). Note that generating and injecting a log statement (*e.g.,* `LoggerUtil.debug("execution ok")`) involves correctly predicting several information: (i) the name of the variable used for the logging (*i.e.,* `LoggerUtil`); (ii) the log level (*i.e.,* `debug`); (iii) the log message (*i.e.,* `"execution ok"`); and (iv) the position in the method in which the log statement must be injected. Thus, when a prediction is generated, three scenarios are possible:

Correct prediction: A prediction that correctly captures all above-described information, *i.e.,* it matches the name used for the variable, the log level, message, and position as written by the original developers.

Partially correct prediction: A prediction that correctly captures a subset of the needed information (*e.g.,* it correctly generates the log statement but injects it in the wrong position).

Wrong prediction: None of the above-described information is correctly predicted.

We answer RQ_1 through the following combination of quantitative and qualitative analysis. On the quantitative side, we report for both LEONID and LANCE the percentage of correct, partially correct, and wrong predictions. For the partially correct, we report the percentage of cases in which each of the "log statement components" (*i.e.,* variable name, log level, log message, and log position) has been correctly predicted. As for the percentage of correct and partially correct predictions, we pairwise compare them among the experimented techniques, using the McNemar's test [McN47], which is a proportion test suitable to pairwise compare dichotomous results of two different treatments. We complement the McNemar's test with the Odds Ratio (OR) effect size. We use the Holm's correction procedure [Hol79] to account for multiple comparisons.

Concerning the quality of the log messages generated by the two techniques, looking for exact matches (*i.e.,* cases in which the generated log message is identical to the one written

by developers) is quite limitative considering that a prediction including a message different but semantically equivalent to the target one could still be valuable. For this reason, we also compute the following four metrics used in Natural Language Processing (NLP) for the assessment of automatically generated text:

BLEU [PRWZ02] assesses the quality of the automatically generated text in terms of n-grams overlap with respect to the target text. The BLEU score ranges between 0 (the sequences are completely different) and 1 (the sequences are identical) and can be computed considering four different values of n (*i.e.,* BLEU-{1, 2, 3, 4}). Besides these four variants, we also compute their geometric mean (*i.e.,* BLEU-A).

METEOR [BL05] is a metric based on the harmonic mean of unigram precision and recall. Compared to BLEU, METEOR uses stemming and synonyms matching to better reflect the human perception of sentences with similar meanings. Values range from 0 to 1, with 1 being a perfect match.

ROUGE [Lin04] is a set of metrics focusing on automatic summarization tasks. We use the ROUGE-LCS (Longest Common Subsequence) variant which returns three values: the recall computed as *LCS(X,Y)/length(X)*, the precision computed as *LCS(X,Y)/length(Y)*, and the F-measure computed as the harmonic mean of recall and precision, where X and Y represent two sequences of tokens.

LEVENSHTEIN Distance [Lev66] provides an indication of the percentage of words that must be changed in the synthesized log message to match the target log message. This is accomplished by computing the normalized token-level Levenshtein distance [Lev66] (NTLev) between the predicted log message and the target one. Such a metric can act as a proxy to estimate the effort required to a developer in fixing a non-perfect log message suggested by the model.

We also statistically compare the distribution of the BLEU-4 (computed at sentence level), METEOR, ROUGE, and LEVENSHTEIN distance related to the predictions generated by LEONID and LANCE. We assume a significance level of 95% and use the Wilcoxon signed-rank test [Wil45], adjusting p-values using the Holm's correction [Hol79]. The Cliff's Delta (d) is used as effect size [GK05] and it is considered: negligible for $|d| < 0.10$, small for $0.10 \leq |d| < 0.33$, medium for $0.33 \leq |d| < 0.474$, and large for $|d| \geq 0.474$ [GK05].

On the qualitative side, we manually inspected 300 of the partially correct predictions generated by both techniques and having all information but the log message correctly predicted. The goal of the inspection is to verify whether the generated log message, while different from the target one, is semantically equivalent to it. To this aim, two of the authors independently inspected all 600 log messages (300 for each approach), with ∼11% (70) arisen conflicts being solved by a third author. We report the percentage of "wrong" log messages generated by both techniques classified as semantically equivalent to the target one.

To answer RQ_2 and evaluate the extent to which LEONID is able to correctly inject multiple log statements, we run LEONID against the test set reported in Table 6.2 (see row *Fine-tuning: Multi-log Injection with IR*). We then report the percentage of correct predictions generated by the approach (*i.e.,* methods for which all n log statements that LEONID was supposed to generate and inject have been correctly predicted). In this case we do not

compute the partially correct predictions since, if a prediction is not completely correct, it is not possible to match the generated log statements with the target ones to compare them. To make this concept more clear, consider the case in which LEONID was asked to generate two log statements s_1 and s_2 but it only injects one statement s_i, being different from both s_1 and s_2. We cannot know whether s_i should be compared with s_1 or with s_2 to assess the percentage of partially correct predictions in terms of *e.g.,* log level. For this reason, we only focus on the predictions being 100% correct (*i.e.,* the output method is identical to the target one).

To answer RQ_3, we run LEONID against the test sets presented in Table 6.3, reporting the confusion matrix of the generated predictions and the corresponding accuracy, recall, and precision. We compare these results with those of: (i) an *optimistic* classifier always predicting *true* (*i.e.,* the method is in need for log statements); (ii) a *pessimistic* classifier always predicting *false* (*i.e.,* no need for log statements); and (iii) a random classifier, randomly predicting *true* or *false* for each input instance. We use the same statistical analysis described for RQ_1 to compare LEONID with the baselines.

6.3 Results Discussion

We discuss the achieved results by research question.

6.3.1 RQ_1: Injecting a single log statement

Table 6.6 reports the results achieved by LEONID and LANCE, in terms of correct and partially correct predictions for the task of single-log injection. For LEONID we only report the results when $k = 5$, since this is the variant that achieved the best performance (results with $k = 1$ and $k = 3$ are available in [Mas23]). The first row of Table 6.6 shows the percentage of correct predictions by both approaches, which is slightly higher for LEONID (+1.8% of relative improvement, from 26.78% to 27.26%). This difference is statistically significant (adj. p-value < 0.01) with 1.12 higher odds of obtaining a correct prediction from LEONID as compared to LANCE.

Table 6.6. RQ_1: Correct and partially correct predictions by LEONID and LANCE on the single-log injection task

Variable	Level	Message	Position	LEONID (k=5)	LANCE	p-value	OR
✓	✓	✓	✓	27.26%	26.78%	<0.01	1.12
✓	-	-	-	76.45%	77.15%	<0.01	0.88
-	✓	-	-	73.53%	74.18%	<0.01	0.91
-	-	✓	-	31.55%	30.16%	<0.01	1.36
-	-	-	✓	82.35%	82.28%	0.71	1.01

The four subsequent rows report the cases in which one of the four log-statement components (variable, level, message, and position) was correctly predicted (✓), independently from whether the other three components were correct or not (—). As it can be seen, there is

no significant difference in the prediction of the log position, with both techniques correctly predicting it in ∼82.3% of cases. Differences are observed for the log variable and level in favor of LANCE (+1.0% and +0.9% relative improvement), and for the log message in favor of LEONID (+4.6% relative improvement). The log message is the part for which we observed the highest OR among all comparisons. Considering that the only difference between LEONID and LANCE is the usage of IR, the improvement in the generation of meaningful log messages we targeted has been at least partially achieved. The latter has, however, a small price to pay in the correct prediction of the log variable and level. Still, for these elements LEONID is able to generate a correct prediction in over 73.5% of cases, while the correct generation of the log message still represents the Achilles' heel of these techniques, with 31.55% correct predictions achieved by LEONID. Thus, we believe that improvements on the log message predictions should be favored even at the expense of losing a bit of prediction capabilities on other elements.

Digging further into the quality of the generated log messages, Table 6.7 reports the results computed using the four NLP metrics presented in Section 6.2 for both models (in bold the best results). All metrics suggest that the log messages generated by LEONID are closer to those written by humans. According to our statistical analysis (results in Table 6.8), all these differences are statistically significant (adj. p-value < 0.001) with, however, a negligible effect size.

Table 6.7. RQ_1: Evaluation Metrics on Log Messages: LEONID *vs* LANCE

	LANCE	**LEONID** ($k = 5$)
BLEU-A [PRWZ02]	31.98	**35.36**
BLEU-1	47.30	**50.00**
BLEU-2	36.30	**39.60**
BLEU-3	33.90	**35.00**
BLEU-4	31.40	**32.40**
METEOR [BL05]	58.60	**60.35**
ROUGE-LCS [Lin04]		
$precision$	42.57	**44.68**
$recall$	44.04	**46.01**
$fmeasure$	42.19	**44.33**
LEVENSHTEIN [Lev66]	44.02	**41.85**

Also the result of our manual inspection of 300 partially correct predictions by LEONID and by LANCE point to a similar story: We found 198 of those generated by LEONID (66%) to report the same information of the target log message, despite being semantically different. The remaining 102 (34%) predictions, instead, reported a log message completely different from the target one or not meaningful at all. For LANCE, the number of semantically equivalent log messages is slightly lower — 192 (64%) — but inline with that observed for LEONID. Examples of different but semantically equivalent log messages generated by LEONID are reported in Fig. 6.3. The methods labeled with "Target Java Method" represent the "oracle", namely the log statement that LEONID was supposed to generate. Those instead labeled with "Predicted Method" represents the generated prediction being different from the expected target but, accordingly to our manual analysis, still valid.

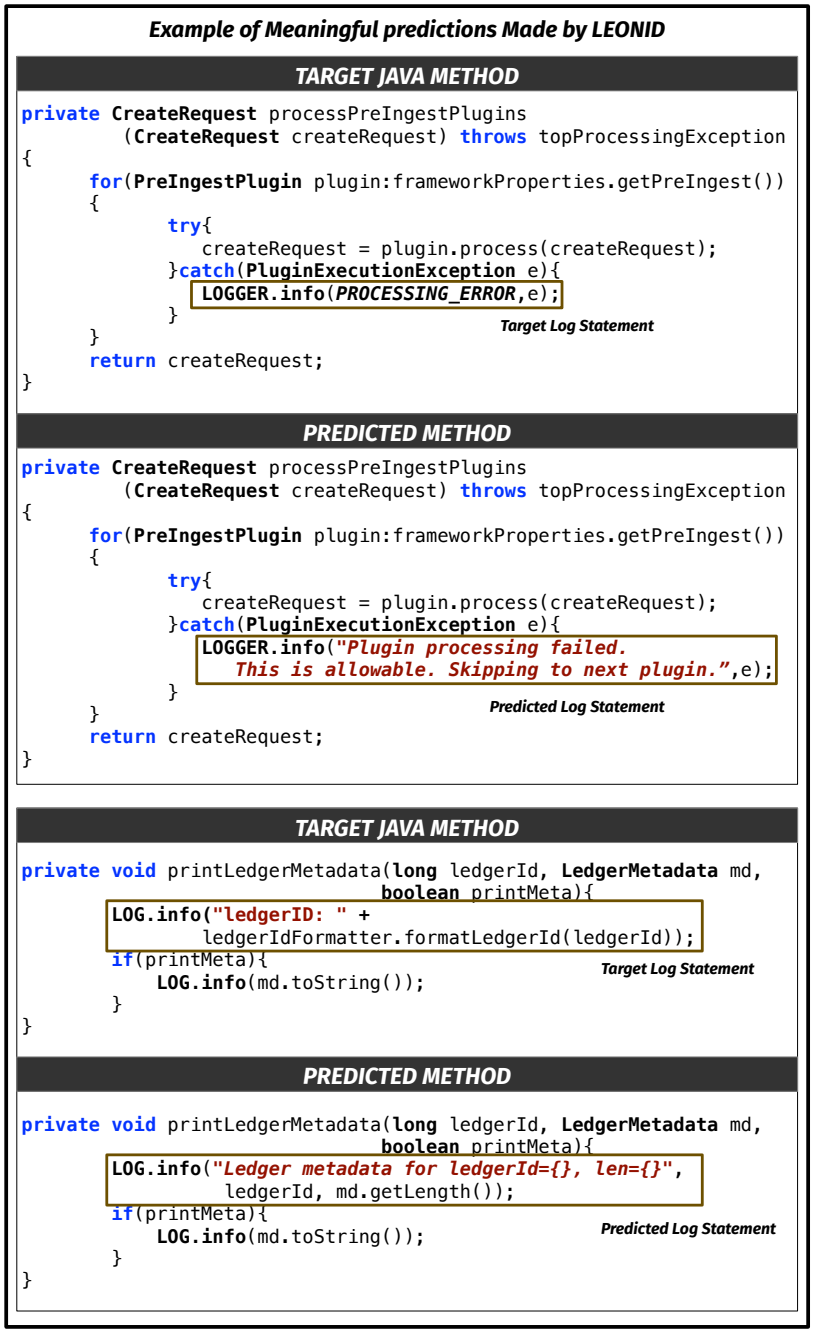


Figure 6.3. Examples of semantically equivalent log messages generated by LEONID

Answer to RQ$_1$. The 3.6 larger training dataset (as compared to the original one we used in [MPB22]), resulted in a boost of performance when predicting the log message (15.20% in [MPB22] *vs* 30.16%). Such a result has been further improved by LEONID, which achieves a +4.6% relative improvement (*i.e.,* 31.55% of correctly generated log messages). All metrics used to assess the quality of the log messages generated by LEONID indicate improvements over LANCE. However, these improvements are marginal, showing that more research is needed to further improve the automated generation of log messages.

Table 6.8. RQ_1: Statistical Tests: LEONID *vs* LANCE for NLP metrics

Comparison	Metric	*p*-value	d
LEONID ($k = 1$) *vs.* LANCE	BLEU-4	<0.001	-0.022 (N)
	METEOR	<0.001	-0.025 (N)
	ROUGE-LCS (f-measure)	<0.001	-0.025 (N)
	LEVENSHTEIN	<0.001	+0.022 (N)
LEONID ($k = 3$) *vs.* LANCE	BLEU-4	<0.001	-0.026 (N)
	METEOR	<0.001	-0.029 (N)
	ROUGE-LCS (f-measure)	<0.001	-0.023 (N)
	LEVENSHTEIN	<0.001	+0.027 (N)
LEONID ($k = 5$) *vs.* LANCE	BLEU-4	<0.001	-0.026 (N)
	METEOR	<0.001	-0.029 (N)
	ROUGE-LCS (f-measure)	<0.001	-0.026 (N)
	LEVENSHTEIN	<0.001	+0.029 (N)

6.3.2 RQ_2: Injecting multiple log statements

As explained in Section 6.2, it is not possible to compute the partially correct predictions in the scenario of multiple log injection. Thus, we limit our discussion to the correct predictions generated by LEONID. Independently from the value of k (*i.e.,* the number of similar coding contexts from which exemplar log messages are extracted), LEONID can correctly predict all log statements to inject in a given method in >23% of cases. Also in this scenario, $k = 5$ is confirmed as the best configuration, with 23.51% of correct predictions. Fig. 6.4 depicts two cases for which LEONID correctly recommended more than one log statement: *four* in ❶ and *three* in ❷.

Interestingly, the drop in performance as compared to the simpler scenario of single log injection is there but is not substantial (27.26% *vs* 23.51%). Remember that in this experiment we removed from a given *Java* method M a random number y of log statements, with $1 \leq y \leq n$ and n being the number of log statements in M. Thus, it is possible that most of the methods in our dataset had $n = 1$ and, as a consequence, $y = 1$ (*i.e.,* LEONID must generate one log statement), thus making the task similar to the single-log injection. For this reason, we inspected our test set and found indeed that 85% of methods in it featured, in their original form, a single log statement. On top of this, there is another 6.7% of methods which originally had more than one log statement and from which we randomly removed $y = 1$ statement, thus again resulting in instances requiring the addition of a single log statement. We clustered the instances in the test set based on the number of log statements that LEONID was required to generate. We created two subsets: (i) *one-log*, having $y = 1$; and (ii) *at-least-two-log*, $y \geq 2$. The *one-log* subset features 91.7% of the instances in the test set (22,104 out of 24,088) and, on those, LEONID achieves 24.1% correct predictions; the *two-log* subset features 1,984 instances (8.3%), on which LEONID has a 17.0% success rate. Thus, there is an actual performance drop when LEONID needs to predict multiple log statements in a given method. Still, in 17% of cases, LEONID is able to inject the same log statements manually written by developers. To give a term of comparison, in our original paper presenting LANCE [MPB22], we reported a 15.2% success rate for the task of single-log injection.

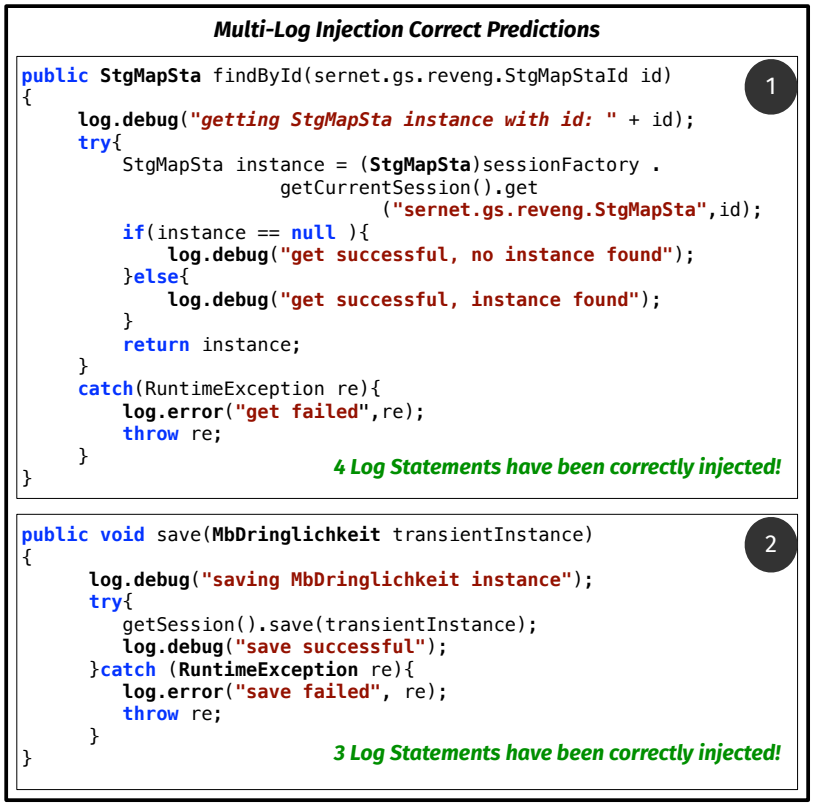


Figure 6.4. Correct predictions made by LEONID when injecting more than one log statement.

Answer to RQ$_2$. LEONID can support the task of multiple log injection, achieving 17.0% of correct predictions when more than one log statement must be injected. It is important to highlight that in this task it is up to the model to infer how many log statements are actually needed in the method given as input, making it more complex than the single-log injection experiment even when only a single log statement must be injected.

6.3.3 RQ$_3$: Deciding whether log statements are needed

Fig. 6.5 reports the confusion matrices for the test sets in Table 6.3, differing for the proportion of *need*/*no need* instances they feature. The rows in the matrices represent the oracle and columns the predictions. For example, the first matrix to the left indicates that out of the 11,627 (11,013+614) methods in *need* for log statements, LEONID correctly identified 11,013 of them, wrongly reporting the remaining 614 as *no need*.

The overall accuracy of the classifier is always very high ($\geq$0.95), indicating that most of instances are correctly classified. Similarly, the recall for the "need" class is always $\geq$0.94 (see Fig. 6.5), suggesting that most of the methods in *need* of log statements are identified.

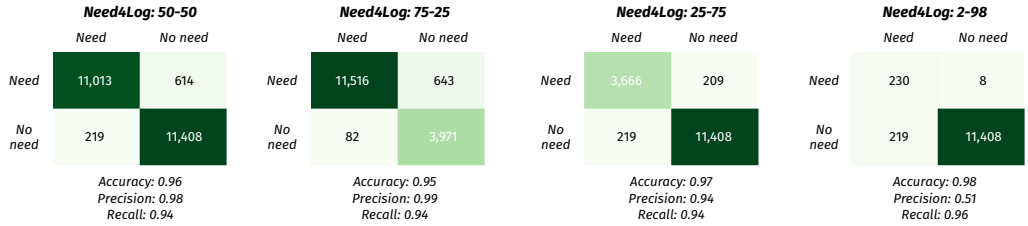


Figure 6.5. RQ_3: Results achieved by LEONID when deciding whether log statements are needed or not in *Java* methods

Instead, the precision drops to 0.51 when the test set is very unbalanced towards the "no need" class, with only 238 *need* instances. Indeed, every classification error weights a lot more on the precision when the number of *need* instances is so low: The 219 misclassifications represent 49% — 219/(230+219) — of the instances that LEONID classifies as in *need* of log statements. Given the overall very good performance achieved by LEONID, we decided to inspect these 219 instances to understand the rationale behind the recommendation by LEONID (*i.e.,* add log statements). What we found is that, indeed, these are cases which are worth the attention of the developers since they may benefit from additional logging.

Fig. 6.6 shows two examples of "*no need* methods" classified by LEONID as in *need* for additional log statements. We added the *LOG_STMT* text bordered in red to indicate positions which may benefit of logging, especially considering the other log statements present in the method. For example, in method `run` ❷ the developers used a log statement to document the reason for the `InterruptedException` in the second `try/catch`, while a similar scenario in the first `try/catch` is not logged. Overall, based on our manual inspection of the "false positives", we are confident that these could still represent valuable recommendations for developers.

When comparing the correct predictions achieved by LEONID with those of the optimistic, pessimistic, and random classifier, we always found a statistically significant difference in favor of LEONID (adj. p-value < 0.001) accompanied by an OR going from a minimum of 6.17 to a maximum of 1,426. The only exception is, as expected, the comparison with the pessimistic classifier on the 2-98 test set, on which the pessimistic classifier achieves 98% of correct predictions. In this case, we found no statistically significant difference (adj. p-value $= 0.63$) with LEONID (detailed results in [Mas23]).

Finally, we conducted a full-system assessment in which we integrated the classifier and generator into a pipeline that first determines whether log statements are necessary, and if so, the module responsible for injecting the logs is activated. Fig. 6.7 provides an overview of how LEONID operates in an `end-to-end` logging scenario. In this context, the `CLASSIFIER` module first determines whether log statements are required for the target method. If log statements are necessary, the `INJECTOR` component inserts one or more log statements into the provided *Java* method.

The achieved results showed that our `end-to-end` logging system can correctly inject ∼23% (5,538/24,088) log statements when needed. This must be compared with the 27.26% achieved in RQ_1 when we only assessed the generation of log statements, "providing" LEONID

Figure 6.6. RQ_3: Examples of methods that may benefit from further logging

only with instances that needed a log statement. Thus, while there is a slight loss in performance, the achieved results confirm the ability of LEONID in automatically assessing the need for log statements.

Answer to RQ_3. LEONID can discriminate between methods *needing* and *not needing* additional log statements, with an accuracy higher than 0.95. This allows LEONID to both predict the need for log statements and generating them.

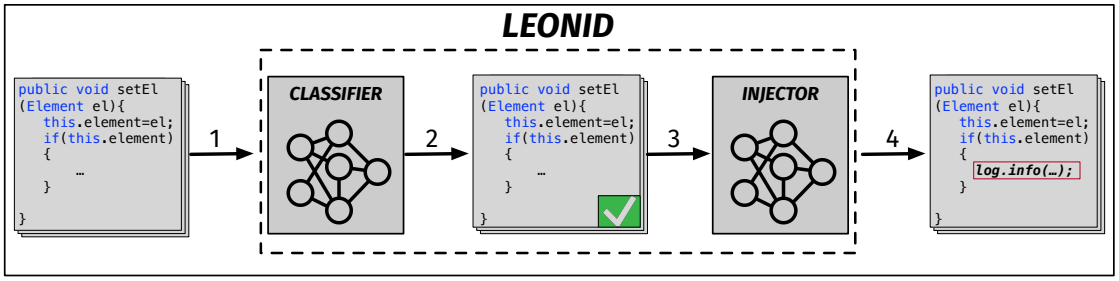


Figure 6.7. RQ_3: Example of LEONID operating in an end-to-end logging scenario (*i.e.,* classification and injection).

6.4 Threats to Validity

Construct validity. The building of our fine-tuning datasets rely on the assumption that the exploited code instances, as written by developers, represent the "correct" predictions that the models should generate. This is especially true for the classifier aimed at predicting whether log statements are needed. For example, the instances that we labeled as "*not needing log statements*" are methods featuring $n \geq 1$ log statements from which we did not remove any log statement. Thus, we assume that these methods need exactly n log statements (*i.e.,* the ones injected by the developers), not one more. This is a strong assumption, as confirmed by the examples in Fig. 6.6.

In addition, there is evidence in the literature showing that some projects may adopt suboptimal logging practices [PFHLN22], thus again posing question on the quality of the adopted ground truth. Future work should involve developers in the assessment of the recommendations generated by LEONID or similar techniques.

Still, using the code written by developers as oracle is a popular practice in DL for SE [TMM+22, TWB+19a, TWB+19b, WTM+20, TDSS22].

It is important to notice that, when preparing the fine-tuning datase we removed log statements from any location within a *Java* method. As a consequence, certain methods may contain empty blocks (*e.g.,* an empty `if` block that only contained the log statemet), thus hinting the model to the right location in which the log statement should be injected (since there is likely something missing in that unusual empty block). To address this problem, we assessed the model's performance on a subset of our initial test set featuring 17,455 instances (∼73% of the original test set) in which there were no empty blocks left within the test method after removing the log statements. The results indicate that LEONID remains competitive even in this more challenging scenario, correctly generating and injecting log statements in 25.30% (4,416/17,455) of the test instances (as compared to the 27.26% obtained on the full test set).

Internal validity. We performed a limited hyperparameters tuning only focused on identifying the best learning rate, while we relied on the best architecture identified by Raffel *et al.* [RSR+20] for the other parameters. We acknowledge that additional tuning can result in improved performance. Also, different similarity measures used to retrieve similar

M_s from the training set may lead to different results. Our choice of the Jaccard similarity was due to practical reasons: Since a given input method to LEONID must be compared with all entries in the training set, we needed a very efficient similarity measure in terms of required computational time. For example, we also implemented a variant of LEONID exploiting CodeBLEU [RGL+20] as a similarity measure. Considering that larger and larger training sets will be likely used in the future, a scalable solution is a must also to make LEONID usable in practice.

External validity. Our research questions have been answered using a dataset being 3.6 times larger as compared to the dataset we originally used when proposing LANCE [MPB22]. Also, the new dataset is more variegated, featuring projects using different build systems (as compared to the Maven-only policy we relied in [MPB22]) and having dependencies towards different logging libraries (differently from the original Log4j-only policy we end up using in [MPB22]). Still, we do not claim generalizability of our findings for different populations of projects, especially those written in other programming languages. This holds not only when looking at the performance achieved on our test set (*i.e.,* different test sets can yield to different results), but also when considering the usage in LEONID of information collected via IR from the training set (*i.e.,* the performance observed for LEONID are bounded to the variety of data present in our training set). Additional experiments are needed to corroborate/contradict our findings.

6.5 Conclusions

We started by discussing the limitations of LANCE [MPB22], the approach we presented at ICSE'22 for the generation of complete log statements. LANCE always assumes that a *single* log statement *must* be injected in a method provided as input. This is a strong assumption considering that a method may not need logging or may need more than one log statement. Thus, we presented LEONID, an extension of LANCE able to partially address these two limitations, making a further step ahed in the automation of logging activities. Also, we experimented in LEONID a combination of DL and IR with the goal of improving the generation of meaningful log messages achieving, however, only limited improvements over LANCE. In light of the results we have obtained, LEONID can ensure up to 27.27% correct predictions, when asked to inject single log statement in *Java* methods. On the other hand, when the model is requested to inject multiple logging statements, we observed that they were correctly added in 17% of the methods. In addition, LEONID is capable of differentiating between methods that necessitate additional log statements and those that do not, achieving an accuracy surpassing 0.95.

Part IV

Code Summarization

The last part of thesis, which comprises four chapters, presents three studies we conducted in the area of code summarization, *i.e.,* creating concise, understandable descriptions of code blocks, functions, or entire programs with the primary goal to provide developers and programmers with insights into the behaviour of the component being summarized.

In the study presented in Chapter 8, we start from the observation that most of the DL-based state-of-the-art techniques for code summarization (*e.g.,* [LJM19, HLX+20b, HLWM20]) train DL models with the aim of learning how to summarize a complete function. This means that, in the case of *Java*, methods are mined from open source projects and linked to the first sentence of their Javadoc to create pairs $<code, description>$ which can be used to train the DL model. However, this level of granularity (*i.e.,* entire functions) might not be ideal for supporting comprehension tasks. Although a developer might grasp the general purpose of a method, they could struggle with understanding certain statements within it. Also, looking at the datasets used in the literature to train these models, we found that the methods' descriptions extracted from the Javadoc are usually very short. For example, the seminal dataset by LeClair and McMillan [LM19], features an average of 7.6 words (median=8.0) to summarize each Java method. While such short descriptions could provide a grasp about the overall goal of the method, it is unlikely that they can actually support a developer struggling to understand it. For this reason, we presented an approach for DL-based snippet-level summarization named STUNT. Creating such a technique presented major challenges, such as building a dataset of $<snippet, description>$ pairs (*i.e.,* a code snippet associated with a natural language description) that, as detailed in Chapter 8, required a major manual effort. The work has been presented in the following publication:

> **Towards Summarizing Code Snippets Using Pre-Trained Transformers**
> Antonio Mastropaolo, Matteo Ciniselli, Luca Pascarella, Rosalia Tufano, Emad Aghajani, Gabriele Bavota. In *In Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension (ICPC 2024)*, pp. *To appear*

Subsequently, we noted that even the most advanced code summarization techniques proposed in the literature still exhibit performance far from the standards required for practical integration into tools for developers. For this reason, we shift the focus on the simpler problem of *code comment completion*, in which the "machine" is in charge of completing a comment that the developer starts writing, similarly to what done for code tokens by code completion techniques [BMM09, RVY14, SLH+21, BAY20]. We present an empirical investigation comparing two techniques for comment completion, namely T5 [RSR+20] and the n-gram model (Chapter 9). The two models are different in nature, with the T5, based on deep learning, exploiting as information to support the completion of a comment C not only the first tokens typed by the developer while writing C but also a "additional context" representing code relevant for C. The n-gram model, instead, does only consider the $n-1$ preceding tokens to predict the n^{th} token. Our results show that the partially written summaries (*i.e.,* the first words written by the developer) help in boosting the performance of the model when automatically summarizing code components. This work has been presented in the following publication:

> **An empirical study on code comment completion**
> Antonio Mastropaolo, Emad Aghajani, Luca Pascarella, Gabriele Bavota. In *In Proceedings of the 37th IEEE International Conference on Software Maintenance and Evolution (ICSME 2021)*, pp. 159-170

Finally, we built on top of our experience in evaluating the performance of code summarization techniques to propose a novel metric aimed at overcoming the limitations of the ones usually adopted in the literature (Chapter 10). Truly determining whether a piece of natural language text (*e.g.,* one automatically generated by a DL model) serves as an effective summary for a code component would require human judgement. However, given the difficulties of running large-scale evaluations with developers, the software engineering community borrowed evaluation metrics from the Natural Language Processing (NLP) field. These include BLEU [PRWZ02], ROUGE [Lin04], and METEOR [BL05]. These metrics have been originally designed to act as a proxy for the quality of automatically generated text (*e.g.,* a translation) by comparing it with a reference (expected) text: The higher the words' overlap between the generated and the reference text, the higher the assessed quality. Particularly, when adopting such metrics for code summarization, the generated summary is contrasted against a single reference text, usually being the original comment written by developers for the code provided as input. However, basing evaluations solely on the similarity or dissimilarity between the generated text and the original summary (*i.e.,* the summary crafted by the developer) proves to be unreliable for two main reasons: (i) it does not account for the suitability of the generated summary in documenting the code, regardless of how the developer originally described it; (ii) various phrasings can express identical meanings, thus, summaries with minimal word overlap can still effectively document the code in the same way. For these reasons, we introduce `SIDE` (**S**ummary al**I**gnment to co**D**e s**E**mantics), a new metric leveraging contrastive learning [SKP15] to model the characteristics of suitable and unsuitable code summaries for a given code. We show that focusing on assessing the suitability of the generated summary for the documented code ignoring the reference summary allows to capture orthogonal aspects of summary quality as compared to state-of-the-art metrics, such as BLEU, ROUGE, METEOR, as well as metrics based on word/sentence embeddings, Also, `SIDE` is the metric having the strongest correlation with human judgment of code summary quality. We also show that `SIDE` can be combined with state-of-the-art metrics to provide a more comprehensive assessment of code summary quality. This work has been presented in the following publication:

> **Evaluating Code Summarization Techniques: A New Metric and an Empirical Characterization**
> Antonio Mastropaolo, Matteo Ciniselli, Massimiliano Di Penta, Gabriele Bavota. In *In Proceedings of the 46th International Conference on Software Engineering (ICSE 2024)*, pp. *To appear*

7

Background and Related Work

We discuss the literature related to snippet-level code summarization, as relevant for Chapter 8, as well as the works dealing with the evaluation of code summarization techniques (relevant for Chapter 10).

7.1 Snippet-level Code Summarization

Iyer *et al.* [IKCZ16] developed CODE-NN, leveraging LSTM networks [HS97] to generate natural language summaries of C# and SQL code snippets. This method was trained and tested on a dataset automatically sourced from Stack Overflow [sta], consisting solely of accepted answers in the form of $\langle Post_{title}, Code_{snippet} \rangle$ pairs. The dataset included 66,015 pairs for C# and 32,337 for SQL. The performance evaluation revealed that when the techniques is inputted with C# code snippets, the presence of descriptive variable names in C#—which closely reflect the code's intent—contributes to superior outcomes compared to generating descriptions for SQL code snippets.

Zheng *et al.* [ZZLW17] developed CODE ATTENTION, an NMT-based approach using RNNs [She18] which incorporates a unique attention module [BCB14] crafted by the researchers to offer more adaptable code token management. This technique was both trained and evaluated on a unique dataset, referred to as "C2CGit", which was automatically constructed from *Java* GitHub repositories. Notably, this dataset is reported to be 20 times larger than the one utilized by Iyer *et al.* [IKCZ16]. The assessments performed on this extensive and more realistic dataset showcased superior outcomes in comparison to the existing state-of-the-art methodologies for code snippet summarization [IKCZ16].

Ye *et al.* [YXZ+20] unveiled CO3, an approach based on Bi-Directional Long Short-Term Memory Network (Bi-LSTM) [HS97] and trained following a dual learning approach [XQC+17]. This entails specializing the model on two distinct tasks: one involves summarizing source code into text, and its counterpart focuses on generating code from textual input. The method underwent training and evaluation on a dataset featuring pairs of Python and SQL code snippets alongside their corresponding descriptions. The outcomes demonstrated that this technique is competitive with the current state-of-the-art benchmarks in the field

[IKCZ16].

Chen *et al.* [CSX+18] developed a novel framework aimed at aligning natural language with source code in a common semantic space for enhanced correlation. The framework makes use of a Bimodal Variational AutoEncoder (BVAE) [KW13] for this purpose and has been proven to surpass CodeNN [IKCZ16] in performance. This was observed after training and testing the suggested technique for generating natural language descriptions for C# and SQL code snippets, using the dataset initially compiled by Iyer *et al.* [IKCZ16].

Huang *et al.* [HHC+20] introduced RL-BLOCKCOM, a technique for code snippet summarization utilizing reinforcement learning for source code documentation. The primary challenge encountered in this work was the development of a training dataset. Indeed, while gathering <*code*, *description*> pairs is straightforward for documentation at the level of entire functions, this becomes significantly more challenging when dealing with snippet of code. This complexity arises for several reasons: (i) not every comment within a function or method qualifies as a description of the code, according to [PB17]; (ii) automatically associating descriptive comments with the corresponding snippet of code is not straightforward, especially when dealing with non-contiguous statements that require human judgment to determine their relevance. For these reasons, Huang *et al.* tried to partially mitigate the issue by exploited an approach proposed by Chen *et al.* [CHL+19a] to automatically detect the scope of code comments. The approach exploits a combination of heuristics and learning-based techniques to automatically identify, given a comment, the set of statements documented by it. Using this approach, Huang *et al.* [HHC+20] built a dataset of ∼124k <*snippet*, *description*> pairs which has been used to train RL-BLOCKCOM, a DL model combining reinforcement learning with a classic encoder-decoder model. RL-BLOCKCOM is able, given a code snippet as input, to automatically document it reaching a BLEU-4 of 24.28.

Although RL-BLOCKCOM sets a new standard in the field of code snippet summarization, outperforming benchmarks previously established [HLX+18b, ABLY18], it faces considerable limitations mainly because of the way its training and testing datasets are compiled, adhering to the approach proposed by Chen *et al.* [CHL+19a] which, as we will show in Chapter 8, results in the construction of a noisy dataset. Also, the authors include comments as short as two words, unlikely to aid in understanding the program.

Contribution in the area. In Chapter 8 we contribute to the area with a manually curated dataset consisting of 6,645 <*snippet*, *description*> pairs, which can be used to train DL-based snippet summarization techniques. We exploit such a dataset to build STUNT, an approach which surpasses the performance of RL-BlockCom, the state-of-the-art technique proposed by Huang *et al.* [HHC+20].

7.2 Evaluation of Code Summarization Techniques and Metrics

We detail studies that focus on investigating factors impacting the soundness of evaluations performed to assess code summarization techniques (Section 7.2.1) as well as metrics capturing code comment quality (Section 7.2.2).

7.2.1 Evaluating Code Summarization Techniques

LeClair and McMillan [LM19] discuss the implications of different preprocessing choices made when preparing the datasets used for the evaluation of code summarization techniques, *e.g.,* splitting the training/test datasets by function *vs* by project. They show that the impact of these choices can be drastic (up to ±33% of performance), and advocate for more standard dataset preparation procedures aimed at increasing results reproducibility.

Stapleton *et al.* [SGL+20] present a study involving 45 university students and professional developers, in which participants were asked to perform program comprehension tasks on functions documented by human-written or automatically generated summaries. They found that participants performed significantly better with human-written summaries. Also, they show that the BLEU and ROUGE metrics computed on the automatically generated summaries do not correlate with the participants' performance in comprehending the documented code. This poses questions on the suitability of these metrics to assess the quality of code summaries, at least when they are used to support program comprehension.

Gros *et al.* [GSDY21] discuss the basic assumption on which several code summarization techniques are built: The code summarization problem resembles the "natural language translation" problem, meaning that it can be seen as a "code-to-natural language translation". The validity of such an assumption would imply the possibility to reuse metrics successfully applied to assess the performance of automated natural language translations (*e.g.,* BLEU) in the context of code summarization. The empirical findings by Gros *et al.* show, however, that the two problems are substantially different. For example, while there is a strong input-output dependence in natural language translation (*i.e.,* similar input sentences result in similar output translations), this is not necessarily the case for code summarization (*i.e.,* similar functions could be commented in completely different ways). Gros *et al.* also show that different BLEU implementations used in the literature provide substantially different results when applied to the same dataset, affecting the ability to compare results reported in different works.

Roy *et al.* [RFA21] focus on the interpretation of metrics used in code summarization, and in particular on BLEU, ROUGE (in several different variants), METEOR, chrF [Pop15], and BERTScore [ZKW+19]. They conducted a study with researchers and practitioners asking them to assess the quality of 36 summaries associated with 6 code snippets (a Java method). Each snippet had six summaries associated, one being the reference summary (*i.e.,* the one written by the original developers) and five resulting from different code summarization techniques [VSP+17, LJM19, HLWM20, ABLY18, XWW+18]. The quality assessment has been performed using an ordinal scale in the range [0, 5] [Opp92] (the higher the better) and focusing on three different aspects of each summary: conciseness, fluency, and content

adequacy. Participants were not aware of which summaries were automatically generated and which, instead, represented the reference summary. Overall, they collected 226 surveys, for a total of 6,253 evaluations (not all participants fully completed the survey). By computing the above-listed evaluation metrics for the generated summaries and comparing them with the human assessment, Roy *et al.* found that metric improvements of less than two points do not indicate any meaningful difference in the quality of the summaries generated by two techniques. For higher differences metrics such as METEOR become reliable proxies for differences in summary quality, while others such as BLEU remain unreliable.

In a subsequent study, Hu *et al.* [HCW+22] confirmed METEOR as the metric having the strongest correlation with the human judgment of code summary quality. Still, the achieved correlation is lower than that observed by human raters. Haque *et al.* [HEBM22] criticize the use of word overlap metrics such as BLEU and ROUGE for assessing the quality of automatically generated summaries. They observe that using word overlap ignores the fact that (i) not all words have the same importance in a sentence; and (ii) the usage of synonyms in the generated summary (*e.g.,* using "delete" instead of "cancel" as in the reference summary) penalizes its quality assessment, which is conceptually wrong. Starting from this observation, they propose the usage of "semantic similarity" metrics based on word/sentence embeddings, which can better capture the similarity between the generated and the reference summaries. They then conducted a study involving 30 professional programmers, each of which was asked to evaluate the quality of a single summary associated with 210 Java methods. Their findings show that embedding-based metrics such as SentenceBERT [RG19] might be more suitable as quality assessment metrics for code summaries as compared to word overlapping metrics such as BLEU. In our study, we also consider all code summary quality metrics used in the work by Haque *et al.*

7.2.2 Assessing the Quality of Code Comments

Several works focused on the automated identification of inconsistent comments, namely comments that are not aligned with the code they document [TYKZ07, TYZ07, TMTL12, LCC+18, WGD+19]. While these techniques are extremely valuable, they are not suited for the quality assessment of an automatically generated summary for the following reasons. First, their output is usually a binary classification indicating whether the comment is consistent or inconsistent, lacking one of the key characteristics of a metric, such as the possibility to compare and rank instances.

Khamis *et al.* [KWR10] propose JavadocMiner, a tool measuring several metrics which the authors consider important for the quality of Javadoc comments, *e.g.,* readability indexes, number of nouns in the comment, whether the Javadoc documents code entities (*e.g.,* parameters, return types) are not implemented in the documented code, etc. This approach exploits Javadoc-specific heuristics which cannot be generalized to the unstructured comments usually generated by code summarization techniques. Also, the quality-related metrics have been defined by the authors and never empirically evaluated against the developers' perception of comment quality.

Steidl *et al.* [SHJ13] propose a machine learning-based approach to classify code com-

ments into seven categories (*i.e.,* copyright, header, member, inline, section, code, and task). Also, they propose a metric named c_coeff to identify low-quality comments. The metric is based on the percentage of words in a comment that is similar to words in the code, where two words are considered similar if they have a Levenshtein distance lower than two (*i.e.,* at most one character must be changed to convert one word into the other).

Contribution in the area. We introduce a novel metric, namely SIDE, which is grounded on contrastive learning and it has been designed to assess whether a textual summary is appropriate for a given code. Our evaluation showed that SIDE can capture a novel dimension of summary quality and better aligns to developers' perception of summary quality as compared to all previously discussed state-of-the-art metrics.

8

Towards Summarizing Code Snippets

Empirical studies showed that code comprehension can take up to 70% of developers' time [MML15, XBL+18]. While code comments can support developers in such a process [dSAdO05], their availability [Spi10] and consistency with the documented code [FWG07, FWGG09, LVLVP15] cannot be taken for granted. A helping hand may come from tools proposed in the literature to automatically document code [RRK15, RJAM17, HAMM10, SPVS11, WLT, WYT13, MAS+13b, SPVS11, MM16, IKCZ16, APS16a, ABLL21, LJM19, HLX+20b, HLWM20, ZWZ+20a]. The most recent techniques (*e.g.,* [LJM19, HLX+20b, HLWM20]) train deep learning (DL) models with the aim of learning how to summarize a given piece of code in natural language. This requires the building of a large-scale dataset composed by pairs $<code, description>$ that can be used to feed the model with $code$ instances asking it to generate their $description$. These approaches are usually trained to work at function-level granularity.

Having such a granularity could be, however, suboptimal to support comprehension activities. For this reason, a few attempts have been made to automatically summarize code snippets rather than entire functions [RRK15, WLT, WYT13, ABLL21, SPVS11, WPVS17, HHC+20]. Most of them are based on information retrieval [RRK15, WLT, WYT13, ABLL21] meaning that, given a code snippet CS to document, the most similar snippet to it is identified in a previously built dataset and its comments are reused to summarize CS. These approaches, while valuable, rely on manually crafted heuristics to automatically identify the "scope of an inner comment", *i.e.,* the statements that a given comment documents. For example, one may assume that an `//inline comment` in Java documents all following statements until a blank line is found [CHL+19b]. As we will show, such a heuristic fails in several cases. Other techniques [SPVS11, WPVS17] exploit pre-defined templates to document code snippets that, however, cannot generalize to all combinations of code statements one could find.

Given the limitations of previous work, Huang *et al.* [HHC+20] proposed an approach exploiting reinforcement learning to document code snippets. The first challenge they faced was the creation of a training dataset. Indeed, while it is relatively easy to collect pairs of $<code, description>$ when working at function-level granularity, this is not the case for code snippets. For this reason, Huang *et al.* exploited an approach proposed by Chen

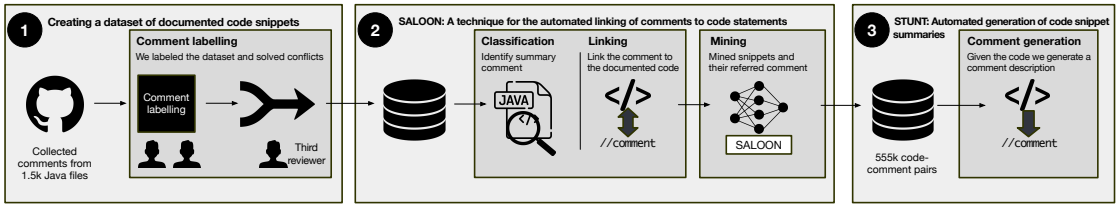


Figure 8.1. Approach Overview

et al. [CHL+19b] to automatically detect the scope of code comments. The approach exploits a combination of heuristics and learning-based techniques to automatically identify, given a comment, the set of statements documented by it. Using this approach, Huang *et al.* [HHC+20] built a dataset of ∼124k <*snippet*, *description*> pairs which has been used to train *RL-BlockCom*, a DL model combining reinforcement learning with a classic encoder-decoder model. *RL-BlockCom* is able, given a code snippet as input, to automatically document it reaching a BLEU-4 [PRWZ02] of 24.28. While being the first DL-based approach to support code snippets' summarization, *RL-BlockCom* suffers of some major limitations mostly related to the way in which its training/test sets have been built exploiting the approach in [CHL+19b]:

1. Simplified/unrealistic linking of code comment to the documented snippet [CHL+19b]. This is due to some of the design choices made in the scope detection approach [CHL+19b]. For example, the authors "*regard the first out-of-scope statement as the demarcation point of the scope of the comment*". This means that, accordingly to their approach, it is not possible for a code comment to document non-contiguous statements. As we will show, our manual validation of 6,645 instances reveals 1598 (∼27%) cases of code comments that document non-contiguous statements. These are all cases which cannot be successfully supported by the scope detection approach and, as a consequence, by *RL-BlockCom*.

2. Lack of filters to identify code summaries [CHL+19b]. Chen *et al.* correctly observed that not all comments "describe" code statements. Thus, they use heuristics to remove commented out code, TODO comments, IDE-generated comments, and non-text comments containing dates or links. Despite these filters, using such an approach to create a training dataset for a snippet summarization approach such as *RL-BlockCom* means feeding it with comments which may not be an actual code summary of the documented snippet. For example, when manually looking at the previously mentioned 6,645 instances, we found 33% of them to just act as a logical split of source code (*i.e.,* a "formatting" comment [PB17]) without providing additional information on the documented code (*e.g.,* a comment `//get messages` put on top of a method call `getMessages()`). These comments are useless to train a code summarizer, but are not excluded from the *RL-BlockCom* training dataset.

3. The training dataset used in RL-BlockCom includes code summaries as short as two words [HHC+20]. These are unlikely to be code summaries useful to support program comprehension.

To address these limitations, in this investigation we take all steps needed to foster the research on snippets summarization, as depicted in Fig. 8.1. First (step 1 in Fig. 8.1), we

manually built a dataset of 6,645 <$snippet$, $description$> pairs, in which we classified the code comment ($description$) as being or not a code summary and linked it to the documented Java statements. Such a dataset has been built by ensuring two evaluators for each analyzed comment, with a third one solving conflicts when needed. The overall effort spent by the six involved authors accounts for 815 man-hours. We use this dataset to fine-tune SALOON (step 2 in Fig. 8.1), a multi-task pre-trained Text-to-Text Transfer Transformer (T5) [RSR+20] model able to take as input an inner comment in a method and (i) classify whether it represents a valid code summary with a 83% accuracy; and (ii) link it to the relevant code snippets it documents with a recall/precision higher than 80%. We show that the performance of SALOON are significantly better than the comment-to-code linking approach by Chen *et al.* [CHL+19b]. Finally (step 3 in Fig. 8.1), we run SALOON on 10k GitHub Java projects to automatically build a large-scale dataset of ∼554k <$snippet$, $description$> pairs. The latter has been used to train and test STUNT, a DL-based approach taking as input a code snippet and automatically generating its code summary. We show that STUNT performs better than IR-based and RL-based baselines *RL-BlockCom*. Despite this finding, our results also show that STUNT is not yet ready to be deployed to developers and point to more research being needed on the task of snippet summarization.

In summary, our contributions are: (i) the largest manually built dataset in the literature featuring classified and linked code comments; (ii) SALOON, a multi-task DL model able to achieve state-of-the-art performance in the tasks of comment classification and linking; and (iii) STUNT, a code snippet summarization model trained on a large-scale and more realistic dataset as compared to the one used in the literature [HHC+20]. The dataset and all code used to train and test the models in this work are available in our replication package [Bli].

8.1 Building a Dataset of Documented Code Snippets

We detail the process used to build a manually validated dataset featuring triplets <D, {CC}, DC> where D represents a natural language comment documenting the code snippet DC (*Documented Code*) and {CC} represents the *Comment Category* (*e.g.,* code summary, TODO comment), with more than one category possibly being assigned to the comment. We later use such a dataset to train and evaluate the model described in Section 8.2, taking as input a comment D and automatically (i) classifying it, thus being able to check whether D is a code summary (*i.e.,* an actual description of the documented code) or another type of comment (*e.g.,* TODOs), and (ii) linking D to the corresponding documented code DC.

8.1.1 Study Design

As a first step to build our dataset we needed to collect the set of code comments D_1, D_2, ..., D_n to manually analyze. To collect these comments, we used the web application by Dabic *et al.* [DAB21] to query GitHub for all Java projects having at least 500 commits, 25 contributors, 10 stars, and not being forks. These filters aim at discarding personal/toy projects and reducing the chance of mining duplicated code. The focus on Java was dictated by the will of accommodating the expertise of the manual validators (*i.e.,* the authors) all having

extensive knowledge of the Java programming language. Despite the focus on Java, our methodology to build the dataset as well as to train the models described in the subsequent sections is general and can be reproduced for different languages.

We randomly cloned 100 of the 1,681 projects resulting from our search on GitHub, for a total of ∼768k Java files.

We parsed their code to identify comments within each method to manually analyze. We ignored Javadoc comments since they document entire methods rather than code snippets: We only considered single-line (starting with "`//`") and multi-line (starting with "`/*`") comments as subject of our manual analysis. Also, we did not extract comments from test methods (*i.e.,* methods annotated with `@Test`) to increase the cohesiveness of our dataset and only focus on documentation related to production code. The manual analysis has been performed by the six authors (from now on, evaluators) through a web app we developed to support the process.

We targeted the labeling of valid comments (*i.e.,* excluding those removed by the above-described procedure) within 1,500 Java files, with the idea of creating a dataset of ∼10k triplets ($<D$, $\{CC\}$, $DC>$). The web app assigned each Java file to two evaluators who independently labeled the comments in it. If the number of comments in a file was higher than 10, the web app randomly selected a number of comments to label going from 10 to m, where m was the actual number of valid comments in the file. Otherwise all comments in the file were labeled. We opted for this process to avoid an evaluator being stuck too much time on a single file. Also, we did not consider comments belonging to methods longer than 1,024 tokens and made sure no duplicated methods were present in the final dataset (*i.e.,* the same method might be present across different files/projects). The filter on the method length was driven by the final usage we envision for our dataset, namely training DL-models which usually works on inputs of limited size (≤512 tokens, or even less, see *e.g.,* [LXH+18, TWB+19a, MPB22, TMM+22, HLWM20, TWB+19b]). Thus, labeling instances longer than 1,024 tokens would have been a waste of resources.

The goal of the labeling was to firstly assign the comment D to one or more categories CCs. The starting set of categories to use was taken from the work by Pascarella *et al.* [PB17] and included: *summary*, *rationale*, *deprecation*, *usage*, *exception*, *TODO*, *incomplete*, *commented code*, *formatter*, and *pointer*. For a detailed overview of all categories, readers are encouraged to consult [PB17]. However, as concrete examples, *summary* represents the classic code description explaining what the code is about, *formatter* is a comment used by developers to better organize the code into logical sections, while *pointer* refers to comments linking external resources. We excluded from the original list by Pascarella *et al.* [PB17] the following categories (i) *directive* and *autogenerated* since, as described by the authors, they both concern comments automatically generated by the IDE; and (ii) *license* and *ownership*, since this information is usually featured in Javadoc comments.

Finally, we merged the *expand* category into *summary*, since the former is defined by the authors as a code description providing more information than a usual summary. Such a distinction is irrelevant for our work. Besides the set of predefined categories, we also gave the possibility to evaluators to define new categories. If an evaluator defined a new category, it was immediately visible to all other evaluators. The following additional categories have

been defined by us: *orphan*, indicating a code comment not linked to any line of code, and *code example*, indicating a comment describing *e.g.,* how to invoke a specific method.

Once the category for a given comment under analysis was defined, the next step was the linking of the comment to the documented code DC. The linking has been performed at line-level granularity. This means, for example, that for a comment D the evaluator could indicate lines 11, 12, and 17 as documented. Note that gaps are possible in DC (*i.e.,* the documented code could be composed by non-contiguous lines). Our replication package [Bli] shows concrete examples of this scenario. Then, we started resolving conflicts arisen from the manual analysis. Two types of conflicts are possible for each manually defined triplet $<D, \{CC\}, DC>$: The two evaluators could have (i) selected a different set $\{CC\}$ when classifying the comment; and (ii) identified different sets of lines (DC) documented by the comment. Out of the 6,645 manually labeled comments, 1,395 (21%) resulted in a conflict: 1,144 were due to different comment categories selected by the evaluators; 47 to differences in the selected DC; 204 concerned both the categories and the DC. Conflicts were solved by a third evaluator not involved in the labeling of the conflicting instance. Overall, we spent 815 man-hours on the labeling and conflict resolution, manually annotating 6,645 comments (with two evaluators for each of them) coming from 1,508 Java files and 85 software projects. We labeled a bit more than the target 1,500 since multiple evaluators were working in parallel without noticing that we hit our target. The obtained dataset, publicly available in our replication package [Bli], is briefly described in the following.

8.1.2 Dataset

Table 8.1. Dataset output of manual labeling

Category	#Instances	Documented Statements		
		mean	median	sd
Summary	3,841	3.40	3.0	2.70
Formatting	2,209	2.32	2.0	2.65
Rationale	983	3.04	2.0	2.74
TODO	258	0.46	0.0	1.16
Commented Code	184	0.00	0.0	0.00
Pointer	33	2.66	2.0	5.27
Orphan	29	0.00	0.0	0.00
Code Example	9	1.77	2.5	1.48
Deprecation	7	3.14	3.0	1.34
Incomplete	2	1.5	1.5	0.70
Overall	**6,645**	**1.83**	**1.60**	**1.80**

Table 8.1 summarizes the dataset obtained as output of our analysis. We excluded from the table the categories for which we did not find any instance (*e.g., exception* [PB17], likely to be more prevalent in Javadoc comments). Since a single comment can be associated to multiple categories (*e.g., summary* and *rationale*), the sum of the "#Instances" column does not add up to the total number of comments we manually classified (*i.e.,* 6,645).

Besides reporting the categories to which the comments in our dataset belong, Table 8.1 also shows descriptive statistics related to the number of statements documented by comments belonging to different categories. As expected, *orphan* and *commented code* comments are not linked to any code statement. More than 80% of *TODO* comments are also not linked to any statement, since in many cases todos are related to *e.g.,* feature that must be implemented. Similarly, the only two *incomplete* comments we found both of them not linked to any code: These are partially written comments needing rework.

The most frequent category is, as expected, the *summary* one (3,841 instances) grouping comments summarizing one or more code statements (on average, 3.40 statements). Another popular category is "*formatting*", with 2,209 instances.

While one could expect no code linked to formatting comments, this is actually not the case since we used such a category also for comments not adding new information to the documented code but just acting as a logical split of the code (*e.g.,* a comment `//get messages` put on top of a method call `getMessages()`).

Finally, comments explaining the *rationale* for implementation choices account for 983 instances. While we focus on the generation of code summaries, these instances often contains interesting information that are hard to automatically synthesize and could represent a seed for future research.

Interestingly, 1,598 of the comments in our dataset (∼27%) include "gaps" in the linked code. This means, for example, that a comment documents lines 11, 12, and 17 (but not lines 13-16) — see [Bli] for concrete examples. This means that approaches to automatically link comment and code must take such a scenario into account. Motivated by these insights, we fill this gap by creating a novel method for classifying and linking code comments, as elucidated in Section 8.2.

8.2 Automatic Classification of Code Comments and Linkage to Documented Code

We start by presenting SALOON (claSsification And Linking Of cOmmeNts), the approach we devised for the classification of code comments and their linking to the documented code (Section 8.2.1). Then, we discuss the design of the study we run to assess its accuracy (Section 8.2.5) and the achieved results (Section 8.2.7).

Once trained, SALOON can be run on hundreds of projects to build a large-scale dataset featuring classified and linked code comments. While we could just refer to SALOON as a "T5 model trained for comment classification and linking", we preferred to name it to simplify the reading when we introduce the other T5 model we train for the task of code summarization (Section 8.3).

8.2.1 Approach Description

SALOON is built on top of T5, a DL transformer-based model [RSR+20] which, in its smaller variant features ∼60M parameters (*i.e.,* $T5_{small}$). To avoid redundancy, we point the reader to Chapter 3 where we extensively discuss the details of the employed model.

8.2.2 Pre-training Dataset

We start from the Java CodeSearchNet dataset [HWG+19], which features ∼1.6M Java methods, ∼499k of which including a Javadoc. Given the tasks we aim at supporting (*i.e.,* automatic classification of code comments and linking to the code they document), there are two "target languages" we aim to expose to T5 during pre-training: Java code and technical natural language in the form of code comments. CodeSearchNet features both of them. We preprocess the dataset by discarding all instances having #tokens > 1,024. During pre-training we treat Java methods and Javadoc comments as separated instances (*i.e.,* we ignore their association), thus removing Java methods and Javadoc comments being longer than 1,024 tokens. Such a filter removed ∼32k instances (*i.e.,* 31,702 methods and 178 Javadoc comments). Then, we excluded instances containing non-ASCII characters as well as Javadoc comments composed by less than 5 tokens (words), since unlikely to represent meaningful code descriptions (∼57k instances removed). After removing duplicates, we end up with 1,870,888 pre-training instances (1,501,013 Java methods and 369,875 Javadoc).

8.2.3 Fine-tuning Dataset

Two fine-tuning datasets are needed to support the tasks we target (*i.e.,* comment classification and linking). For comment classification, we built a dataset composed by pairs $\langle M_{j,D_i}, C_c \rangle$, in which a specific inner comment D_i within a method M_j is linked to a category C_c classifying it (*e.g.,* code summary). For comment-to-code linking, we built a dataset featuring pairs $\langle M_{j,D_i}, DC \rangle$, in which DC reports the M_j's statements documented by D_i. Both datasets have been extracted from the manually built dataset of 6,645 classified and linked comments (Section 8.1).

Comment classification. Given the goal of our work (*i.e.,* summarizing code snippets), we are interested in automatically identifying comments we classified as *code summary* while excluding all the others. Starting from the dataset in Table 8.1, we extracted 3,841 $\langle M_{j,D_i}, C_c \rangle$ having C_c = *code summary* and 2,921 having having C_c = *other*. Basically, we target the training of a binary classifier taking as input a code comment (D_i) in the context of the method it belongs to (M_j) and guessing whether it is a code summary or not.

The specific input we provide to T5 is M_j's code with special tokens `<comment></comment>` surrounding the comment of interest (this is the representation of M_{j,D_i}), and expect as output either "*code summary*" or "*other*" (*i.e.,* C_c).

Differently from the pre-training dataset, we did not need to remove sequences longer than 1,024 tokens, since this has already been done in the first place during the building of the dataset described in Section 8.1. We randomly split the dataset into 80% training, 10% evaluation, and 10% test. The first row in Table 8.2 shows the number of instances in these three sets.

Code Linking. Concerning the task of liking comments to code snippets, our training instances are only those comments that we manually labelled as *code summary*. Indeed, we are interested in linking this specific type of comments to their code. Thus, we start from the 3,841 *code summary* instances to build the needed $\langle M_{j,D_i}, DC \rangle$ pairs. Concerning the representation of M_{j,D_i}, it is similar to the previously discussed for the comment classification

dataset (*i.e.,* the method M_j with special tags surrounding the inner comment of interest D_i) with the only difference being a special tag <N> preceding each statement and reporting its line number in an incremental fashion.

As for the expected output DC (*i.e.,* documented code), it is represented as a stream of "<N>" tags representing the line numbers (*i.e.,* statements) within M_j linked to D_i (*e.g.,* <1><2><4>). Such a representation allows marking non-contiguous statements documented by D_i. The code linking fine-tuning dataset is composed by 3,841 instances split into 80% training, 10% evaluation, and 10% test as shown in the second row of Table 8.2. Note that to ensure a fair evaluation of the proposed approach, we split the dataset by taking into consideration the Java class from which these methods were originally extracted.

Task	Train	Eval	Test
Comment Classification	4,833	726	1,203
Code Linking	2,805	403	633

Table 8.2. Fine-tuning datasets

8.2.4 Training Procedure and Hyperparameters Tuning

We evaluated the performance of eight T5 models (four pre-trained and four non pre-trained) on the evaluation set of each task in terms of correct predictions, namely cases in which the generated output (*i.e.,* the comment category or the documented statements) was identical to the expected output.

We pre-train the T5 model from scratch (*i.e.,* starting from random weights) rather than starting from already pre-trained models for code such as CodeT5 [WWJH21], which is based on the same architecture proposed by Raffel *et al.* [RSR+20] we exploit in our investigation. Our decision is primarily motivated by the desire to have a model pre-trained on a single programming language (Java) as opposed to a multi-language model (as CodeT5).

We pre-train T5 for 300k steps using a 2x2 TPU topology (8 cores) from Google Colab with a batch size of 16. During pre-training, we randomly mask 15% of tokens in an instance (*i.e.,* Java method or Javadoc comment), asking the model to guess the masked tokens. To avoid over-fitting, we monitored the loss function every 10k steps and stopped the training if such value did not improve after 12 consecutive evaluations (*i.e.,* after 120k steps, one epoch on our pre-training dataset). We use the canonical $T5_{small}$ configuration [RSR+20] during pre-training. We also used the pre-training dataset to train a SentencePiece model (*i.e.,* a tokenizer for neural text processing) with vocabulary size set to 32k word pieces.

We fine-tuned a pre-trained and a non pre-trained model experimenting with four different learning rate schedulers (thus leading to eight overall trained models).

Constant Learning Rate (C-LR) fixes the learning rate during the whole training; Inverse Square Root Learning Rate (ISR-LR), in which the learning rate decays as the inverse square root of the training step; Slanted Triangular Learning Rate (ST-LR), in which the learning rate first linearly increases and then linearly decays to the starting value; and Polynomial Decay Learning Rate (PD-LR), having the learning rate decaying polynomially from an initial value

to an ending value in the given decay steps. The parameters used for the learning rates are available in [Bli].

We fine-tuned each of the eight models for a total of 75k steps on the fine-tuning training set of each task. We include in our replication package [Bli] a table showing the percentage of correct predictions (for the *comment classification* task), precision and recall (for the *code linking* task) achieved by each of the pre-trained and non pre-trained models on the evaluation sets.

Overall, the pre-trained models work substantially better, especially when it comes to the *code linking* task. In particular, in their respective best configuration, pre-trained models achieve (i) a 75% classification accuracy in the *comment classification* task as compared to the 58% of the non pre-trained models; and (ii) 85% precision and 89% recall in the *code linking* task, as compared to the 53% precision and 67% recall of the non pre-trained models. Such a result is expected considering that the fine-tuning training datasets are quite small due to the substantial manual effort required to build them (∼6.7k instances for *comment classification* and ∼3.8k for *code linking*). Having small fine-tuning datasets is the scenario in which pre-training is known to bring major benefits [RJ19]. As for the learning rate, the best results are achieved with ISR-LR when pre-training and with PD-LR when not pre-training.

To obtain the final model to use in SALOON, we fine-tuned the best performing model (*i.e.,* pre-trained with ISR-LR) using an early-stopping strategy in which we evaluated the model on the evaluation sets every 5k steps, stopping when no improvements were observed for 5 consecutive evaluations. We discuss the results achieved by SALOON as compared to other baselines in Section 8.2.7.

8.2.5 Study Design

The goal of the study is to assess the accuracy of SALOON in the two tasks it has been trained for: *comment classification* and *code linking*. The context is represented by the test sets reported in Table 8.2, featuring 1,203 instances for the task of comment classification and 633 for the task of code linking.

Concerning the comment classification task, we do not compare SALOON against any baseline, since our goal (*i.e.,* identifying only code summaries) is quite specific of our work. Instead, we compare the performance of SALOON against the three following baselines for the task of code linking (the implementation of all baselines is publicly available [Bli]).

Heuristic-1: blank line [CHL$^+$19b]. The first baseline is a straightforward heuristic assuming that a given `//inline comment` documents all following statements until a blank line is reached.

Heuristic-2: token-based string similarity [FWG07]. The basic idea of this heuristic is that statements sharing terms with a code comment are more likely to be documented by it. We use the token-based string similarity by Fluri *et al.* [FWG07] to compute the textual similarity between each comment in the test set and all statements in the method it belongs to. A statement is linked to the comment if its similarity with it is higher or equal than a threshold λ. The similarity is computed as the percentage of overlapping terms between the two strings (*i.e.,* comment and statement), with the terms being extracted through space

splitting. We experiment with different values for λ, going from 0.1 (*i.e.,* 10% of terms are shared between the two strings) to 0.9 at steps of 0.1.

ML-based solution [CHL$^+$19b]. The approach by Chen *et al.* [CHL$^+$19b] relies on the random forest machine learning algorithm to classify statements in a method as linked or not to a given comment. Unfortunately, the source code of such approach is not available and, thus, we had to reimplement it following the description in the corresponding article. In a nutshell, the approach works as follows. The random forest uses three families of features to characterize a given statement and classify it as linked or not to a given comment. The first family comprises eight "code features", capturing characteristics of the statement, such as the statement type (*e.g.,* `if`, `for`) and whether the statement shares method calls with the statements preceding and following it. The second family includes four "comment features", focusing on characteristics of the comment of interest, such as its length and the number of verbs/nouns it contains. Finally, the third family groups four "relationship features", representing the relationship between the comment and the statement (*e.g.,* textual similarity). For a fair comparison, we train the random forest on the same training set used for SALOON.

8.2.6 Data Collection And Analysis

Concerning the *comment classification* task, we run SALOON on the test set and report the accuracy of the model in classifying comments representing "code summaries". As for the *code linking*, we start computing the percentage of **correct predictions**, namely cases in which all statements linked to a comment in the test set match the ones in the oracle. This means that a comment instance correctly linked to two out of the three statements it documents is considered wrong. We also compute the **recall** and **precision** of the techniques at statement-level. The recall is computed as TP/(TP+FN), where TP represents the set of code-to-comment links correctly identified by a technique (*i.e.,* a statement correctly linked to a comment) and FN are the set of correct code-to-comment links in the oracle missed by the approach. The precision is instead computed as TP/(TP+FP), with FP representing the code-to-comment links wrongly reported by the approach (*i.e.,* statements wrongly identified as linked to the comment). We also statistically compare the techniques assuming a significance level of 95%. We compare precision and recall using the Wilcoxon signed-rank test [Wil45]. To control for multiple pairwise comparisons (*e.g.,* SALOON's precision compared with that of the three baselines), we adjust p-values with Holm's correction [Hol79].

We estimate the magnitude of the differences using the Cliff's Delta (d), a non-parametric effect size measure [GK05]. We follow well-established guidelines to interpret the effect size: negligible for $|d| < 0.10$, small for $0.10 \leq |d| < 0.33$, medium for $0.33 \leq |d| < 0.474$, and large for $|d| \geq 0.474$ [GK05]. As for the percentage of correct predictions, we pairwise compare them among the experimented techniques, using the McNemar's test [McN47], which is a proportion test suitable to pairwise compare dichotomous results of two different treatments. We complement the McNemar's test with the Odds Ratio (OR) effect size. Also in this case we use the Holm's correction procedure [Hol79] to account for multiple comparisons.

8.2.7 Results Discussion

As for the *comment classification* task, SALOON correctly classifies 78.05% (939/1,203) of instances. Out of the 633 *code summary* comments present in the test set, 536 (84%) have been correctly classified, while 97 have been mistakenly reported as *other*.

Concerning the 570 "*other*" comments, SALOON correctly predicted 403 (70%) of them, wrongly reporting 167 instances as *code summary*. This results in a recall=0.85 and precision=0.76 when identifying a comment as a *code summary*. This means that by running our approach on the comments of a previously unseen software system, we can expect to identify 85% of code summaries present in it accompanied, however, by 25% of false positives (*i.e.,* non *code summary* comments).

Table 8.3. T5 *vs* baselines on the code linking task

Technique	Correct Predictions	Recall	Precision
Blank line [CHL+19b]	0.20	0.87	0.57
Token-based similarity [FWG07]			
λ=0.1	0.03	0.62	0.33
λ=0.2	0.05	0.38	0.34
λ=0.3	0.05	0.23	0.26
ML-based [CHL+19b]	0.23	0.49	0.58
SALOON	**0.58**	**0.89**	**0.86**

Concerning the *code linking* task, Table 8.3 reports the correct predictions (*i.e.,* for a given comment in our test set **all** linked statements have been correctly identified), recall, and precision achieved by SALOON and the three baselines. Table 8.4 reports the results of the statistical tests. For the Cliff's Delta d we use N, S, M, and L to indicate its magnitude from Negligible to Large.

Note that for the *token-based string similarity* baseline we report the results achieved with different values of λ (*i.e.,* minimum similarity threshold to link a code statement to a comment).

While we also experimented with values going up to 0.9 [Bli], the recall values were too close to 0 to consider these variants as reasonable baselines.

SALOON predicts all statements linked to a given comment in 58% of cases, against the 23% achieved by the best-performing baseline (*ML-based*). The *blank-line technique* achieves 20% of correct predictions.

The results of the statistical tests confirm the better performance ensured by SALOON in terms of correct predictions: McNemar's test always indicates significant differences in terms of correct predictions accompanied by ORs indicating that SALOON has between 15.80 to 70.80 higher odds of providing a correct prediction against the baselines.

Recall and precision values confirm the superiority of SALOON for the *code linking* task. In terms of recall, SALOON is able to correctly link 89% of statements in our dataset, achieving the best performance among all the experimented techniques. While the *blank-line* approach achieves a similar recall (87%) it pays a much higher price in terms of precision, with a 43% false positive rates as compared to the 14% of SALOON. Note that a high recall

Table 8.4. Code linking task: SALOON *vs* baselines

Comparison	Metric	*p*-value	d	OR
Blank line [CHL+19b] *vs* SALOON	Correct Predictions	<0.05	-	19.28
	Recall	<0.05	-0.04 (N)	-
	Precision	<0.05	-0.48 (L)	-
Token sim.(0.1) [FWG07] *vs* SALOON	Correct Predictions	<0.05	-	70.80
	Recall	<0.05	-0.45 (M)	-
	Precision	<0.05	-0.75 (L)	-
Token sim.(0.2) [FWG07] *vs* SALOON	Correct Predictions	<0.05	-	37.77
	Recall	<0.05	-0.66 (L)	-
	Precision	<0.05	-0.68 (L)	-
Token sim.(0.3) [FWG07] *vs* SALOON	Correct Predictions	<0.05	-	38.00
	Recall	<0.05	-0.80(L)	-
	Precision	<0.05	-0.73 (L)	-
ML-Based [CHL+19b] *vs* SALOON	Correct Predictions	<0.05	-	15.80
	Recall	<0.05	-0.49 (L)	-
	Precision	<0.05	-0.33 (M)	-

for this heuristic is expected, considering that it links all statements following a comment until a blank line is found. The *ML-Based* technique can only predict half of the correct links (0.49) while achieving a precision score of 0.58. Accordingly to our results, the *token-based similarity* heuristic does not represent a viable solution for the *code linking* task: The best results are achieved when considering (λ=0.1) as a threshold, for which the technique can ensure a recall of 0.62 and a precision of 0.33. Differences in terms of recall and precision are always statistical significant (see Table 8.4). The effect size is in most of cases medium or large, with the only exception of the recall test comparing T5 with the *blank-line* baseline, for which a negligible effect size is reported.

To summarize, SALOON is able to identify comments representing code summaries with a recall of 0.85 and a precision of 0.76. Also, it achieves state-of-the-art results in linking comments to the documented code, with a recall of 0.89 and a precision of 0.86. In Section 8.3 we explain how we exploit this model to build a large-scale dataset aimed at training a T5 fine-tuned for the task of code snippet summarization.

8.3 Snippets Summarization Using T5

We discuss how we trained a T5 model for the task of code snippet summarization (Section 8.3.1), the study we run to evaluate it (Section 8.3.4) and the achieved results (Section 8.3.6). We refer to the snippet summarization approach as “STUNT” (SnippeT sUmmarizatioN using T5).

8.3.1 Approach Description

We rely on the same T5 architecture anticipated in Section 8.2.1 and we reuse the same pre-trained model we built for the *comment classification* and *code linking* tasks. Indeed, as explained in Section 8.2.2, we pre-trained the model on a dataset composed by ∼1.5M

Java methods and their inner comments and ∼370k Javadoc comments. Thus, T5 has been pre-trained to acquire knowledge about the two "target languages" relevant for the summarization task as well (*i.e.,* Java code and technical language used to summarize it). We detail the fine-tuning dataset and the training procedure.

8.3.2 Fine-tuning Dataset

We used the GHS tool by Dabic *et al.* [DAB21] to query GitHub for all public non-forked Java projects with minimum 50 commits, 5 contributors, and 10 stars. The idea of these filters was to remove toy/personal projects while still obtaining a large set of projects to provide as input to SALOON with the goal of identifying comments representing summaries and linking them to the relevant code. We cloned 10k of the 18.7k projects returned by our query and extracted their methods using srcML [CDM13].

We excluded all methods longer than 512 tokens and removed all duplicates, obtaining a set of methods S. We also removed duplicates between our pre-training dataset and S and between our manually labeled dataset (Section 8.1.2) and S.

Concerning the removal of duplicates between the pre-training dataset and S, this was needed since S is our starting point to build the fine-tuning dataset for the snippet summarization task from which we will also extract the test set on which STUNT will be evaluated. Thus, we ensure that STUNT is not evaluated on already seen instances. As for the removal of duplicates between the manually labeled dataset and S, this is due to the fact that SALOON (*i.e.,* our approach for comment classification and linking) has been trained on those instances and we will run it on S to build the fine-tuning dataset for STUNT (*i.e.,* for code summarization). Running SALOON on already seen instances would inflate its performance, and not provide a realistic picture of what can be achieved by training STUNT on a dataset automatically built using SALOON.

From the remaining methods, we extracted all inner comments, filtering out those shorter than 5 words (unlikely to represent a meaningful code summary). As done in previous code summarization works [LJM19], we lowercased and stemmed the comments (using the spaCy NLP library [spa]). Then, for each comment D_i extracted from a method M_j we created an instance M_{j,D_i} in which M_j's code features special tokens `<comment></comment>` to surround the comment of interest (D_i). This means that if M_j features three inner comments, three M_{j,D_i} instances will be created, each having a different comment (D_i) "tagged". This format is the one expected by SALOON to automatically (i) classify D_i as *code summary* or *other*, and (ii) link D_i to the relevant code statements.

The above-described process resulted in 2,210,602 M_{j,D_i} instances that we provided as input to SALOON, which classified 907,660 of them as *code summary*. Among these, SALOON automatically linked code statements to the *code summaries* in ∼85% of cases (776,531). These instances are $\langle M_{j,DC}, D_i \rangle$ pairs, where $M_{j,DC}$ represents the method M_j with special tokens `<start><end>` surrounding the statements (DC) documented by D_i.

If more non-contiguous statements are documented, multiple `<start><end>` pairs are injected in M_j. These pairs are those needed to fine-tune STUNT for the task of snippet summarization: the input provided to the model is $M_{j,DC}$ (*i.e.,* a snippet to document) and

the expected output is the documentation D_i. To avoid favoring the model during testing, we also removed all duplicates at snippet-level granularity. This means that if we have in our dataset two different methods containing the same *DC* (*i.e.,* the same code snippet to document), we only keep one of them. Also, being SALOON an automated approach, it is expected to produce wrong instances (*e.g.,* comments linked to wrong statements) which, in turn, will penalize the performance of STUNT. By manually inspecting a sample of the pairs in our dataset, we noticed that one clear case of wrong instances are those in which the model had very low confidence in identifying the documented statements thus producing random symbols rather than the expected documented line numbers. We automatically remove those instances, obtaining a set of 554,748 pairs, split into 80% training (443,798), 10% evaluation (55,475), and 10% testing (55,475).

8.3.3 Training Procedure and Hyperparameters Tuning

As explained, we started from the already pre-trained T5 model. We then followed the same hyperparameters tuning discussed in Section 8.2.4, assessing the performance of four different learning scheduler on the evaluation set using the BLEU-4 score [PRWZ02] as performance metric. The BLEU-4 variant computes the BLEU score by considering the overlap of 4-grams between the generated text (*i.e.,* the synthesized snippet summary) and the target text (*i.e.,* the summary written by the original developers). This metric has been used by most of the previous work on code summarization (see *e.g.,* [HLWM20, ACRC20, WZS+22, LHWM20, LBM21, LJM19, HLX+18a, HHC+20, HLX+20b, HLX+18b, IKCZ16, WZY+18, WSD+21, WLL+20, YXZ+20, ZWZ+20b]). Each of the four models has been trained for 100k steps before its evaluation. C-LR (*i.e.,* constant learning rate) provided the best performance. Data about this evaluation are available in our replication package [Bli].

Once identified the best T5 variant, we fine-tuned it for up to 500k steps, using an early-stopping strategy to tame over-fitting. To this aim, we monitored the BLEU-4 score achieved on the evaluation set every 5k steps, stopping the training when no improvements were observed after 5 consecutive evaluations.

8.3.4 Study Design

The goal is to assess the accuracy of STUNT for snippet summarization. The context is represented by (i) 55,475 $\langle M_{j,DC}, D_i \rangle$ pairs identified by SALOON as described in Section 8.3.2 and belonging to the test set, and (ii) the test set made publicly available by Huang *et al.* [HHC+20] when presenting *RL-BlockCom*, the state-of-the-art snippet summarization approach discussed at the beginning of this Chapter.

We assess the performance of STUNT against an information retrieval (IR)-based technique (*i.e.,* IR-Jaccard) and *RL-BlockCom*. To explain the basic idea behind the IR-based baseline let us remind that both our training and test set are composed by $\langle M_{j,DC}, D_i \rangle$ pairs. Given a pair in the test set, the baseline retrieves in the training set the pair having the *DC* snippet being the most similar to the one in the test set pair. This means that this pair contains a documented snippet that is very similar to the one in the test set for which we have to generate a code summary. Once identified the most similar snippet in the training set,

the IR-based technique reuses its description to document the instance in the test set. This baseline serves as a representative of works using IR to retrieve similar comments from a given dataset, including *e.g.,* [WYT13].

IR: Jaccard index [Han04]. IR-Jaccard identifies the most similar snippet using the Jaccard similarity index. The latter considers the overlapping between two sets of unique elements, representing in our case the tokens composing the documented code (DC) in the test instance and in each of the training instances. Indeed, we need to compare each instance in the test set to all those in the training set to find the most similar one. The similarity is computed as the percentage of overlapping tokens between the two sets.

An additional baseline for STUNT is *RL-BlockCom* by Huang *et al.* [HHC+20]. Despite the code being available, we did not manage to re-train their approach on our dataset. We contacted the authors asking for help without, however, receiving answer. Thus, as an alternative form of comparison, we thought about training and testing STUNT on their dataset, which is publicly available, and then comparing the summaries generated by STUNT with those generated by *RL-BlockCom*. Unfortunately, the authors did not make the summaries generated by their approach publicly available. The only viable form of comparison we found was to (i) re-train STUNT on the training dataset made available by Huang *et al.* [HHC+20] and used to train *RL-BlockCom*; (ii) use this trained version of STUNT to generate predictions on the same test set on which *RL-BlockCom* has been evaluated; (iii) use the evaluation scripts made available by Huang *et al.* for the computation of the sentence-level BLEU score; and (iv) compare the achieved results with those reported in their paper. Indeed, not having access to the summaries generated by *RL-BlockCom* does not allow us to double-check the data reported in the original paper nor to compute additional metrics besides those used by the authors (BLEU). Note also that the training/test datasets shared by Huang *et al.* feature pairs $\langle DC,\ D_i \rangle$ as compared to our $\langle M_{j,DC},\ D_i \rangle$ pairs. This means that STUNT cannot exploit the contextual information of the method M_j when generating the predictions on their dataset.

8.3.5 Data Collection And Analysis

To compare the performance of our model against the two IR-based baselines, we exploit three metrics explained in the following.

Out of those, only BLEU has been used in the comparison with *RL-BlockCom* for the reasons previously explained.

BLEU [PRWZ02] assesses the quality of the automatically generated summaries by assigning a score between 0 and 1. In our case, 1 indicates that the natural language summary automatically generated is identical to the one originally written by the developer. Since in the test set we built there are no summaries shorter than 4 words, we use the BLEU-4 variant in the comparison with the IR-based baselines. When comparing with *RL-BlockCom* on their test set, we also compute BLEU-1, BLEU-2 and BLEU-3 as done by Huang *et al.* [HHC+20].

METEOR [BL05] is a metric based on the harmonic mean of unigram precision and recall (the recall is weighted higher than the precision). Compared to BLEU, METEOR uses stemming and synonyms matching to better match the human perception of sentences with

similar meanings. Values range from 0 to 1, with 1 being a perfect match.

ROUGE [Lin04] is a set of metrics focusing on automatic summarization tasks. We use the ROUGE-LCS (Longest Common Subsequence) variant, which identifies longest co-occurring in sequence n-grams. ROUGE-LCS returns three values, the recall computed as *LCS(X,Y)/length(X)*, the precision computed as *LCS(X,Y)/length(Y)*, and the F-measure computed as the harmonic mean of recall and precision where X and Y represent two sequences of tokens.

We also statistically compare the different approaches assuming a significance level of 95%. Also in this case we use the Wilcoxon signed-rank test [Wil45], adjusting p-values to account for multiple comparisons (Holm's correction procedure [Hol79]) and the Cliff's Delta (d) as effect size measure [GK05]. The statistical comparison was not possible with *RL-BlockCom* since we only had access to the overall BLEU scores reported in the paper (*i.e.,* the BLEU scores for each generated summary were not available).

8.3.6 Results

Table 8.5 compares STUNT and *RL-BlockCom*, using the values reported in the paper by Huang *et al.* [HHC+20] as BLEU scores for *RL-BlockCom*. STUNT achieves better performance for all BLEU scores, outperforming the state-of-the-art approach by a large margin (*e.g.,* +7 points of BLEU-4). A deeper comparison of the two techniques is not possible since the summaries generated by *RL-BlockCom* are not available.

	RL-Com	**STUNT**
BLEU-1	32.18	**34.17**
BLEU-2	25.98	**31.09**
BLEU-3	24.36	**30.63**
BLEU-4	24.28	**31.22**

Table 8.5. BLEU scores: STUNT *vs* RL-BlockCom [HHC+20]

Table 8.6 compares STUNT against IR-Jaccard on the large-scale dataset we built. Accordingly to all metrics used in our evaluation, the gap in performance between STUNT and the baseline (*i.e.,* IR-Jaccard) is substantial, with at least a +11 in terms of BLEU-4, a +12 in terms of ROUGE-LCS f-measure, and a +16 in terms of METEOR score. As observed by Roy *et al.* [RFA21], METEOR is "*extremely reliable for differences greater than 2 points*" in assessing code summarization quality as perceived by humans (*i.e.,* also humans are likely to prefer STUNT's summaries over those generated by the baselines).

The statistical analyses presented in Table 8.7 validate STUNT's superior performance compared to IR-Jaccard. Notably, we observe significant p-values and medium effect sizes for BLEU-4 and ROUGE-LCS (f-measure), while METEOR demonstrates a large effect size.

While the metrics we computed provide a fair comparison among the experimented techniques, they do not give a clear idea of the quality of the summaries generated by STUNT. To this aim two of the authors manually inspected 384 randomly selected summaries generated by STUNT for which the generated text was different from the target summary (*i.e.,* the one written by developers). These are cases that in a "binary quantitative evaluation" would

	IR-Jaccard	STUNT
BLEU-4 [PRWZ02]	27.43	**38.42**
ROUGE-LCS [Lin04]		
precision	23.00	**34.21**
recall	23.04	**37.39**
f measure	22.33	**34.57**
METEOR [BL05]	25.04	**41.75**

Table 8.6. Evaluation Metrics: STUNT *vs* IR-Jaccard

Table 8.7. Statistical Tests: STUNT *vs* IR-Jaccard

Comparison	Metric	*p*-value	d
IR (Jaccard) *vs* STUNT	BLEU-4	<0.001	-0.451 (M)
	ROUGE-LCS (f-measure)	<0.001	-0.471 (M)
	METEOR	<0.001	-0.474 (L)

be classified as wrong predictions. The authors independently classified each summary as *meaningful* or *not meaningful*, based on the ability of the summary to properly describe the documented snippet. In the labeling, the two involved authors achieved a Cohen's kappa [Coh60] of 0.61, indicating a substantial agreement when measuring inter-rater reliability for categorical items.

Conflicts, arisen in 71 cases and have been solved through open discussion among the authors. We classified 224 summaries as meaningful, with some of them representing even a better summary than the one manually written by the original developers. For example, we found the comment `if we have a frontend then we need to get the action list` to be more meaningful and detailed than the `exit if we do not have a frontend` written the developer. However, we also want to highlight the ∼41% (160) of automatically generated summaries which were not meaningful and that stress how far we still are from obtaining a code summarizer being accurate enough to be deployed to developers (*i.e.,* generating correct summaries in most of cases).

8.4 Threats to Validity

We discuss the threats that could affect the validity of our findings.

Internal Validity. Building our dataset of classified and linked code comments (Section 8.1) involved a certain degree of subjectivity. To partially address this threat, two evaluators independently assessed each instance and a third one solved conflicts when needed. Still, imprecisions are possible.

We performed a limited hyperparameters tuning of the T5 models, only experimenting with different learning rates. For example, we did not change the number of layers, but relied on the default $T5_{small}$ architecture by Raffel *et al.* [RSR+20]. Better results could be achieved with additional tuning. Also, relying on pre-trained code models like CodeT5 [WWJH21], might produce better results.

Construct Validity. When experimenting with SALOON, we compared its performance with the technique by Chen *et al.* [CHL+19b]. However, since their approach is not publicly available, we had to reimplement it following the paper's description.

We release our implementation [Bli]. Still related to the used baselines, as explained in Section 8.3.4 we did not manage to compare STUNT (our approach for snippet summarization) with RL-BlockCom [HHC+20] on our dataset. At least, we presented a comparison performed on the dataset released by the authors.

External Validity. The manually built dataset represents the obvious bottleneck in terms of generalizability, since it is based on the analysis of "only" 1,508 Java files and also capped our training/evaluation of SALOON. Still, building such a dataset costed over 815 man-hours. Also, we did not compare our technique against general purpose large language models such as ChatGPT [cha], since designing a fair evaluation is challenging due to the unknown training set behind these LLMs. For example, we could have tested the ability of ChatGPT to summarize specific snippets which, however, were part of its training set together with their related comment.

8.5 Conclusions

We targeted the problem of code snippet summarization, presenting (i) a manually labeled dataset of ∼6.6k code comments classified in terms of information they provide (*e.g.,* code summary) and linked to the code statements they document; (ii) SALOON, a T5 model trained on our manually built dataset to automatically classify and link inner comments in Java code; and (iii) STUNT, a T5 model trained on a large-scale dataset of documented code snippets automatically created by running SALOON on 10k Java projects.

We achieved promising results for both code linking and snippet summarization, pointing however to the need for research in this field. Our dataset and our models, publicly released [Bli], represent a step in that direction.

9

Supporting Code Summarization via Comment Completion Techniques

Despite the substantial improvements brought by DL techniques in addressing the *code comment generation* problem, the findings reported even in the most recent empirical studies show how these techniques are still far from being useful tools for software developers. For example, in Chapter 3 and Chapter 8 we have shown that state-of-the-art techniques struggle to generate comments equivalent to those written by humans. In this chapter, we tackle the simpler problem of *code comment completion*, in which the "machine" is in charge of completing a comment that the developer starts writing, similarly to what done for code tokens by code completion techniques [BMM09, RVY14, SLH+21, BAY20, CCP+21].

The code comment completion problem has been firstly tackled by Ciurumelea *et al.* [CPG20] in the context of Python code: They study whether a deep learning model can predict the next word that a developer is likely to type while commenting code. This is, to the best of our knowledge, the only work done in this area. Stemming from their idea, we present in this thesis a large-scale study assessing the ability of a simple *n*-gram model and of T5 [RSR+20] in supporting code comment completion for Java programs. As compared to the work by Ciurumelea *et al.* [CPG20], besides focusing on a different context (*i.e.,* Python *vs* Java) we: (i) investigate the actual advantages brought by a DL-based model (T5) over a simpler *n*-gram model that can be trained in a fraction of the time required by T5; (ii) do not limit our study to predicting the single next word the developer is likely to type, but evaluate how the investigated techniques perform when asked to predict longer word sequences (*e.g.,* the next 10 words), providing a more advanced completion support to developers. We also study the complementarity of the two techniques and report qualitative examples of correct and wrong predictions to understand their strengths and limitations.

Our study has been run on a dataset composed by 497,328 Java methods with their related comments. The achieved results can be summarized as follows. First, the T5 model outperforms the *n*-gram model, achieving superior performance in all the comment completion scenario we tested. Second, despite being more performant, the T5 model exploits as input not only the first part of the comment already written by the developer (also used by the *n*-gram model), but also a *context* representing the relevant code for the comment

to complete. This means that the T5 model, as we tested it, can only be used when the developer writes the comment after the code has been already implemented (the assumption made by approaches for automated code documentation [HLX+18a, IKCZ16, APS16b, HLX+20b, HLWM20]). Thus, the applicability of the *n*-gram model is higher (*i.e.,* it can be used also when the code is not yet implemented).

9.1 T5 to Support Code Comment Completion

In this investigation, we leverage a T5 architecture (*i.e.,* $T5_{small}$) which has been detailed extensively in Chapter 3. Therefore, we choose not to recount those specifics, and instead begin by outlining how we configure the model to facilitate the task in question (*i.e.,* code comment completion)

9.1.1 Problem Definition

We instantiate the T5 to the problem of code comment completion in Java. We tackle the problem at method-level granularity, meaning that we expect the model to learn how to autocomplete a code comment used to document a method or part of it. In Java, a method can be documented using a Javadoc comment (that we indicate with C_{JD}) or inner comments (C_I). Each comment is relevant to a specific *context*. For example, the context of a C_{JD} is the entire method, while the context of an C_I can be a single line or a code block.

Given a *context* and an incomplete comment (either a C_{JD} or a C_I), the trained model must predict the tokens needed to complete the comment. This implies that we must build a training dataset in which code comments are linked to the relevant part of the code they document (*i.e.,* the context). While this is trivial for C_{JD} comments, a heuristic is needed for C_I comments, since they are not explicitly linked to certain statements. Section 9.1.2 describes how we built such a dataset.

We pre-train the T5 by randomly masking tokens in comments asking the model to guess the masked tokens (Section 9.1.3). This builds the shared knowledge that we then specialize in two fine-tuning tasks, namely $Inner\text{-}comment_{task}$ and $Javadoc_{task}$, consisting in predicting the missing part of inner and Javadoc comments, respectively (Section 9.1.4).

9.1.2 Dataset Preparation

We start from the *CodeSearchNet* dataset [HWG+19], providing 6M functions from open-source projects. We only focus on the Java subset, composed of ∼1.5M methods. We extract the set of instances using the `docstring` (*i.e.,* the method's *Javadoc*) and `function` fields. Then, we run a pre-processing aimed at preparing our dataset.

First, we discarded instances having $\#tokens \geq 256$, where $\#tokens = \#function\_tokens + \#docstring\_tokens$ (*i.e., #tokens* is the total number of tokens used to represent both the method and the comments associated to it). Such a filter is needed to limit the computational expense of training the model, and removed ∼9% of instances from the dataset. Then, we excluded instances containing non ASCII characters as well as comments composed by less than

three tokens (words), since unlikely to represent an interesting scenario for code comment completion. We also excluded comments representing instances of self-admitted technical debt (SATD) [PS14a] (*i.e.,* comments documenting temporary workaround). Such a choice was dictated by the fact that we are interested in training our model to complete comments describing a method (or part of it) rather than comments used to document technical debt and very likely to be project-specific. We adopted a simple heuristic to discard SATD comments, excluding all comments starting with `TOFIX`, `TODO`, and `FIXME`. We are aware that more complex state-of-the-art techniques for SATD detection could be used (see *e.g.,* [RXX+19a]), but we preferred a simpler unsupervised heuristic for our pre-processing pipeline.

We discarded commented code statements using the *codetype* library [cod]. Then, we cleaned comments by replacing links with a special *_LINK_* token (using the *urlextract* library [url]), dates with a *_NUM_* token (using the *datefinder* library [dat]), and references to code components with a *_REF_* token. The latter are only handled in Javadoc comments exploiting the `@link` tag used to reference code elements. We further clean Javadoc comments by stripping *HTML/XML* tags using the *BeautifulSoup* library [bso].

Then, we removed from each method all C_I (inner comments) that are "*orphans*", *i.e.,* C_I followed **and** preceded by at least one blank line. As previously explained, to train the T5, we need to link each comment to its *context*.

Javadoc comments are linked to the whole method, while for inner comments we adopt a heuristic that would not work with orphan comments (*i.e.,* we cannot know what lines of code they likely document) — details in Section 9.1.4. As a last step in the processing of inner comments, we merge in a single C_I subsequent inline comments that are not interleaved by empty lines or code statements. This is done since they likely represent a single multi-line comment.

Type of instance	**#Instances Pre-training**	**#Instances Fine-tuning**
Inner comment(s) only **(D1)**	45,764	33,590
Javadoc comment only **(D2)**	232,121	115,904
Javadoc & Inner comments **(D3)**	53,667	16,282
Total	331,552	165,776

Table 9.1. Instances used for pre-training and fine-tuning.

Finally, we removed duplicates, obtaining the study dataset composed of 497,328 instances, with each instance being a method with its related Javadoc and/or inner comments.

We randomly split this dataset using 2/3 of it for pre-training and 1/3 for fine-tuning. Table 9.1 shows the number of instances in each of the two datasets, distinguishing between instances only containing inner comments (**D1**), only containing Javadoc (**D2**), and featuring both (**D3**).

9.1.3 Pre-training of T5

In the *pre-training* phase, we use a self-supervised task similar to the one used by Raffel *et al.* [RSR+20], consisting of masking tokens in natural language sentences and asking the model to guess the masked tokens. Since we want the model to learn how to generate comments given a certain context, we randomly mask 15% of tokens appearing in the comment-related part of each instance (Javadoc or inner comments). Tokens representing the method code were not masked. The pre-training has been performed for 200k steps.

We also created a new *SentencePiece* [KR18] (*i.e.,* a tokenizer for neural text processing) model by training it on the entire pre-training dataset, in such a way that it can handle both code and comments. We set its length to 32k wordpieces.

9.1.4 Fine-tuning of T5

Once pre-trained, we fine-tune the T5 model in a multi-task setting, in which the two tasks are represented in our case by the automatic completion of (i) Javadoc comments and (ii) inner comments. Having a single model fine-tuned on these two strongly related tasks could result in an effective transfer learning, in which knowledge gained by the model while learning a task (*e.g., Javadoc*$_{task}$) can be transferred to other similar tasks (*e.g., Inner-comment*$_{task}$).

9.1.5 Preparing the Dataset for the Model Fine-Tuning

We further process the 165,776 instances selected for the fine-tuning (see Table 9.1) through the following steps.

Processing Javadoc instances (datasets D2 and D3 in Table 9.1). For the *Javadoc*$_{task}$ we assume that the *context* documented by a C_{JD} is represented by the whole method.

Thus, given an instance composed by {`C`$_{\texttt{JD}}$, `context`} (*i.e.,* a Javadoc comment followed by the method it documents) we simply swap the position of the two elements and add a special separation token <sep> to delimit the comment, obtaining an instance in the form: {`context`<sep>`C`$_{\texttt{JD}}$<sep>}. The rationale behind this transformation is to "force" the model to process the context before predicting the missing parts of the comment.

Once this is done, we use the method `sent_tokenize` from the *nltk* library [nlt] to split the C_{JD} in each instance into the y sentences composing it. Then, we take the first sentence s_1 and remove the remaining $y-1$. Assuming s_1 is composed by n tokens, we randomly extract five different integers between 1 and $n-1$, and use them to create five variants of s_1 each one having the last $n-m_i$ tokens masked, where m_i is one of the five random integers.

By training the model on these five masked sentences, the learning is focused on guessing how to finalize an incomplete sentence in a comment the developer is writing. Let us justify and explain this process. First, we remove the $y-1$ following sentences because we assume that a developer writes the comment linearly, starting from the first to the last sentence. Thus, when the developer is writing the first sentence, the remaining $y-1$ do not exist yet. Second, at most $n-1$ tokens can be masked in a sentence composed by n tokens, since at least the first token must be provided by the developer (otherwise, the task would be comment generation rather than completion). Third, the choice of creating five different variants for

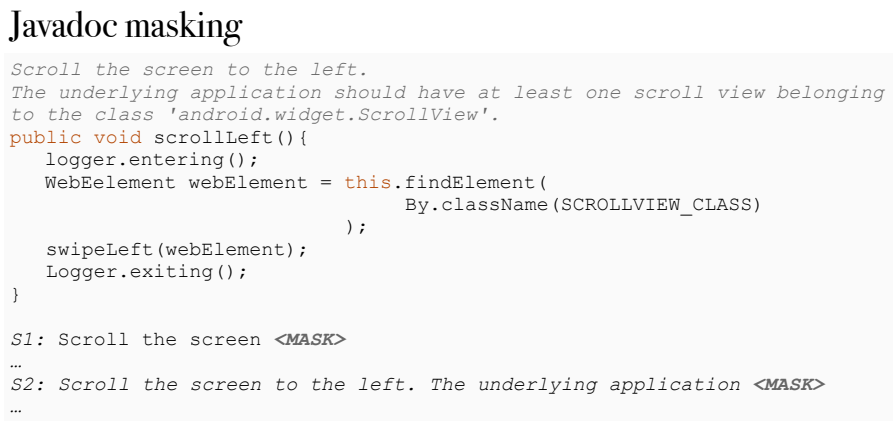


Figure 9.1. Example of the Javadoc masking process

a given sentence is a tradeoff between experimenting with a different number of masked tokens for each sentence and considering all possible combinations of masked tokens, that would lead to an excessive number of training instances.

Such a process is repeated for all y sentences composing the C_{JD}, hiding the sentences following the one under process while keeping the ones preceding it. Thus, for each instance in our fine-tuning dataset, we create up to y*5 instances (*i.e.,* y sentences with five different sets of masked tokens). Less than five instances are created if C_{JD} has less than six tokens, since it would not be possible to mask five different sets of tokens (excluding the first one). Fig. 9.1 shows an example of masking performed on a single instance. The original instance is reported on top of the figure, and two sentences compose its C_{JD}. This results in the creation of 10 fine-tuning instances. For improved readability, Fig. 9.1 only shows one of the instances generated for each sentence.

Processing Inner-comment instances (datasets D1 and D3 in Table 9.1). For the *Inner-comment*$_{task}$ the first step to perform when preparing the fine-tuning dataset is the identification of the *context* relevant for a given inner method. We define the following heuristic to identify, given an C_I in a specific instance, the context that can help the model in understanding how to support C_I completion. We use Fig. 9.2 as a running example to explain the heuristic, showing an example of instance having a single C_I. Starting from the line l_{C_I} containing it, we expand the context both above and below it until specific conditions are met.

In particular, while expanding above and below l_{C_I}, we stop when we find one of the following: (i) an empty line, (ii) a closing curly brace, or (iii) another code comment. In both cases, we do not expand the context out of the method (*e.g.,* if none of the above conditions is met while expanding above l_{C_I}, we stop at the method signature). In the example shown in Fig. 9.2, there is no above context since we immediately hit an empty line, while the context below is stopped when we find the first closed curly brace. It is important to clarify that the *context* we identify is not necessarily the part of code documented by C_I. However, our interest is to provide the model with the relevant code surrounding C_I, to allow it exploiting

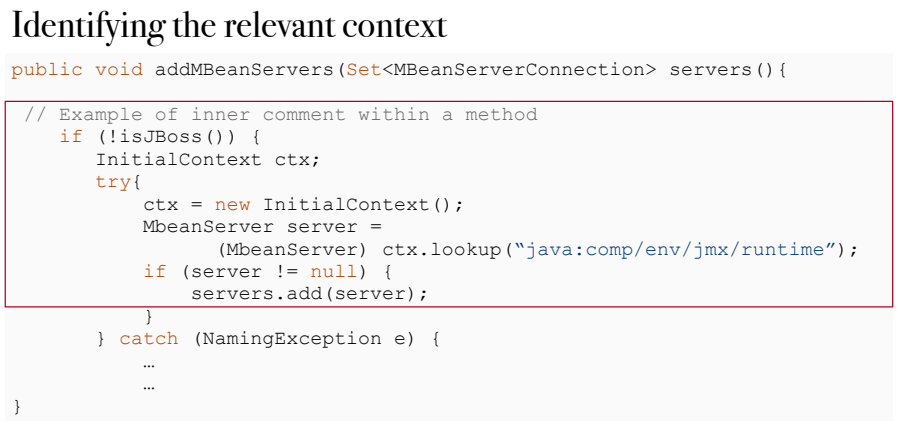


Figure 9.2. Identification of relevant context of an inner comment

useful information to support the comment completion. Once linked each C_I to its context, we perform the same processing previously described for the C_{JD} instances (*i.e.,* splitting the comment into sentences and creating five variants of each sentence, each having a different number of tokens masked at the end of it).

Data sources	**Train**	**Eval**	**Test**
$Javadoc_{task}$	1,398,135	174,624	175,084
$Inner\text{-}comment_{task}$	272,944	34,705	34,138
Total	1,671,079	209,329	209,222

Table 9.2. Instances used for the fine-tuning

9.1.6 Dataset Splitting

Table 9.2 shows the fine-tuning dataset we obtained from the above-described process. We split it into 80%-10%-10% for train, test and validation, respectively. The dataset for the $Javadoc_{task}$ dominates, in terms of size, the one for the $Inner\text{-}comment_{task}$.

This could result in an unbalanced effectiveness of the model for the two tasks. In other words, the model could perform better on the $Javadoc_{task}$ and being less effective on $Inner\text{-}comment_{task}$. However, as pointed out by Arivazhagan *et al.* [ABF+19], there is no free lunch in choosing the balancing strategy when training a multi-task model, with each strategy having its pros and cons (*e.g.,* oversampling of less represented datasets negatively impacts the performance of the most representative task). For this reason, we decided not to perform any particular adaptation of our training set but to follow the true data distribution when creating batches.

9.1.7 Decoding Strategy

The given output layer's values allow different possible decoding strategies to generate the output token streams. We adopt a *greedy decoding* strategy since it we believe it is better suited for the problem we are tackling. Indeed, it provides the developer with the most likely completion (rather than with a set of completions as, for example, a *beam search* would do. Indeed, with multiple options a developer would likely spend more time selecting among different tool suggestions rather than writing the comment themself.

9.1.8 Hyperparameter Tuning

We rely on the configurations used by Mastropaolo *et al.* [MSC+21]. Concerning the pre-training, we do not tune the hyperparameters of the T5 model because the pre-training step is task-agnostic, and this would provide limited benefits. Instead, we experiment with four different learning rates schedule for the fine-tuning phase, using the configurations reported in Table 9.3.

Learning Rate Type	**Parameters**
Constant (C-LR)	$LR = 0.001$
Slanted Triangular (ST-LR)	$LR_{starting} = 0.001$
	$LR_{max} = 0.01$
	$Ratio = 32$
	$Cut = 0.1$
Inverse Square Root (ISQ-LR)	$LR_{starting} = 0.01$
	$Warmup = 10,000$
Polynomial Decay (PD-LR)	$LR_{starting} = 0.1$
	$LR_{end} = 0.01$
	$Power = 0.5$

Table 9.3. Learning-rates tested for hyperparameter tuning

We fine-tune the model for 100k steps for each configuration; then, we compute the percentage of perfect predictions (*i.e.,* cases in which the model can correctly predict all masked tokens in the comment) achieved on both tasks in the evaluation set. The achieved results reported in Table 9.4 showed a slight superiority of the *slanted triangular* (column 2) that we use in our study.

Task	**C-LR**	**ST-LR**	**ISQ-LR**	**PD-LR**
$Javadoc_{task}$	30.77%	33.60%	31.73%	30.65%
$Inner\text{-}comment_{task}$	9.38%	10.55%	9.70%	9.51%

Table 9.4. Hyperparameter tuning results

9.2 Study Design

The *goal* of this study is to experiment the extent to which a T5 model and an *n*-gram model can help developers in writing comments faster by supporting code comment completion.

In particular, we answer the following research question: *To what extent can T5 and n-gram models be leveraged to support the auto-completion of code comments?*

We answer this research question by using the 209,222 test set instances in Table 9.2 as a context for our study. This means that the two trained models (*i.e.,* T5 and *n*-gram model) are run on the same test set instances to predict the masked parts of code comments. The T5 model has been trained as described in Section 9.1, while in the following we explain how we implemented and trained the *n*-gram models. The code and data used in our study are publicly available [repe].

9.2.1 N-Gram Model

An *n*-gram model can predict a single token following the $n-1$ tokens preceding it. We publicly release the implementation of our *n*-gram model in Python [repe]. We trained the *n*-gram model by using the same instances used to fine-tune the T5 without, however, the masked tokens. We experimented with three different values for n (*i.e.,* $n = 3$, $n = 5$ and $n = 7$). Even though the *n*-gram model is meant to predict a single token given the $n-1$ preceding tokens, we designed a fair comparison for the scenario in which we mask more than one token. In particular, we use the *n*-gram model in the following way: Let us assume that we are predicting, using a 3-gram model, how to complete a sentence having five tokens $\texttt{T}$, of which the last two are masked ($\texttt{M}$): <$\texttt{T}_1$, $\texttt{T}_2$, $\texttt{T}_3$, $\texttt{M}_4$, $\texttt{M}_5$>. We provide as input to the model $\texttt{T}_2$ and $\texttt{T}_3$ to predict $\texttt{M}_4$, obtaining the model prediction $\texttt{P}_4$. Then, we use $\texttt{T}_3$ and $\texttt{P}_4$ to predict $\texttt{M}_5$, thus obtaining the predicted sentence <$\texttt{T}_1$, $\texttt{T}_2$, $\texttt{T}_3$, $\texttt{P}_4$, $\texttt{P}_5$>. Basically, all predictions are joined to predict multiple contiguous tokens.

We experimented with the different values of *n* by running the models on the evaluation set, taking the best one (*i.e., 5*-gram) and comparing its performance with that of the T5 model. We report the results achieved for the *3*-gram and *7*-gram models in our replication package [repe].

9.2.2 Evaluation Metrics and Data Analysis

We compare the T5 and 5-gram models using six metrics.

Perfect predictions: This metric measures the percentage of cases (*i.e.,* instances in the test set) in which the sequence predicted by the model equals the oracle sequence. Since we want to investigate the extent to which the experimented techniques can actually support comment completion, we compute the perfect predictions when the technique is only required to guess the first masked token, the first two, the first three, etc. For example, if we assume that in the previous prediction <$\texttt{T}_1$, $\texttt{T}_2$, $\texttt{T}_3$, $\texttt{P}_4$, $\texttt{P}_5$> $\texttt{P}_4$ is correct while $\texttt{P}_5$ is wrong, we will consider this as a perfect prediction only when looking at the first token to predict, and as a wrong one when looking at the first two.

We compute the percentage of perfect predictions when trying to predict the first k masked tokens, with k going from 1 to 10 at steps of 1 (*i.e.,* 1, 2, 3, etc.) and for the most challenging scenario in which $k > 10$ (*i.e.,* more than 10 masked tokens must be correctly predicted to consider this prediction as a perfect one).

BLEU score [PRWZ02]: This metric measures how similar the candidate (predicted) and reference (oracle) texts are. Given a size n, the candidate and reference texts are broken into n-grams, and the algorithm determines how many n-grams of the candidate text appear in the reference text. The BLEU score ranges between 0 (the sequences are completely different) and 1 (the sequences are identical). For both the tasks, we compute the BLEU-{1, 2, 3, 4} and their geometric mean (*i.e.,* BLEU-A). Due to the way in which the BLEU-X is computed (*i.e.,* at least X tokens must be part of the prediction task) we only compute the BLEU-A metric when the number of tokens for a given prediction is at least 4. As done for perfect predictions, we report results for different values of k.

Levenshtein distance [Lev66]: To understand the effort needed by developers to convert a prediction generated by the model into a correct comment, we compute the Levenshtein distance at word-level, *i.e.,* the minimum number of word edits (insertions, deletions or substitutions) needed to convert the predicted comment into the reference one. Also in this case, we report results for different values of k.

Overlap metrics: We also compute the complementarity between the T5 and the n-gram model. Let PP_{T5_t} and PP_{NG_t} be the sets of perfect predictions achieved by T5 and the n-gram model, where $t \in \{Inner\text{-}comment_{task}, Javadoc_{task}\}$. We compute the following metrics:

$$Shared_t = \frac{|PP_{T5_t} \cap PP_{NG_t}|}{|PP_{T5_t} \cup PP_{NG_t}|}$$

$$OnlyT5_t = \frac{|PP_{T5_t} \setminus PP_{NG_t}|}{|PP_{T5_t} \cup PP_{NG_t}|} \qquad OnlyNG_t = \frac{|PP_{NG_t} \setminus PP_{T5_t}|}{|PP_{T5_t} \cup PP_{NG_t}|}$$

$Shared_t$ measures the percentage of perfect predictions shared between the two compared approaches, while $OnlyT5_t$ and $OnlyNG_t$ measure the percentage of cases in which the perfect prediction is only achieved by T5 or the n-gram model, respectively, on the task t.

Confidence Analysis: Both models provide, together with the generated prediction, a score between 0 and 1 indicating the confidence of the prediction, with 1 being the maximum confidence. We check whether the *confidence* of the predictions can be used as a reliable proxy for their "quality". If this is the case then, a possible implementation of these models into a tool could take advantage of this proxy to automatically filter out low-confidence predictions. For each model, we compute the confidence level for the two sets of "perfect" and wrong predictions comparing their average confidence.

Qualitative analysis of the predictions: To better understand the strengths and weaknesses of the models, we analyze more closely the generated predictions. First, we check what type of words on two models are able to correctly predict.

We do this by performing a *Part-of-Speech (POS) TAG* analysis. For each test set instance we check the POS category of each masked word using 12 POS categories [PDM12] (*e.g.,* adjective, adverb, noun). Then, for each POS category, we compute for both models the

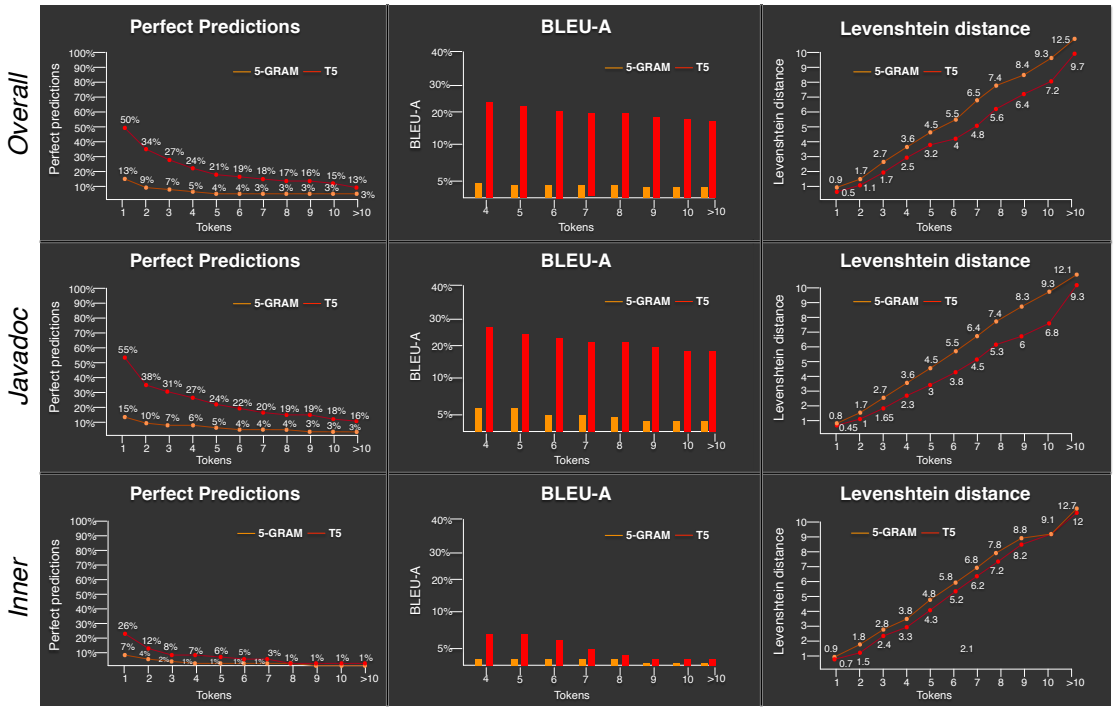


Figure 9.3. Performance of the T5 model against the 5-gram model

percentage of times they were able to correctly predict it. Such an analysis is useful to understand whether the words correctly predicted by the models are mostly trivial ones (*e.g.,* determiners such as "the", "a") or also more challenging words representing, for example, nouns. On top of that, we discuss examples of predictions made by the two models.

A statistical comparison between the T5 and the *n*-gram model is performed using the McNemar's test [McN47] and Odds Ratios (ORs) on the perfect predictions they can generate.

9.3 Results Discussion

Fig. 9.3 depicts the results achieved by the T5 and 5-gram model in terms of perfect predictions, BLEU-A score, and Levenshtein distance computed for predictions of different lengths (k). The results for the other metrics (*e.g.,* BLEU-1 to BLEU-4) can be found in our replication package [repe]. The middle and bottom parts of Fig. 9.3 show the results achieved in the $Javadoc_{task}$ and $Inner\text{-}comment_{task}$, respectively, while the top part aggregates the results of the two datasets.

In both the evaluated tasks (*i.e.,* $Javadoc_{task}$ and $Inner\text{-}comment_{task}$) as well as overall, T5 outperforms the 5-gram model by a significant span for all metrics considered in our study. This is especially true when the two models are required to predict a limited number of tokens ($k \leq 6$) following the ones already written by the developer in the code com-

ment. For example, when only the subsequent word must be predicted (*i.e.,* $k = 1$), T5 can achieve more than 50% of perfect predictions in the $Javadoc_{task}$ and more than 25% in the $Inner\text{-}comment_{task}$. The 5-gram model, in both scenarios, achieves less than 16% of perfect predictions.

The difference in performance is particularly remarkable for the $Javadoc_{task}$, in which the T5 model achieves better results compared to the $Inner\text{-}comment_{task}$. Such a finding might be due to two important factors. First, as explained in Section 9.1.6, the fine-tuning dataset used for the $Javadoc_{task}$ is larger than the one used for the $Inner\text{-}comment_{task}$, thus likely providing more knowledge to the model about the vocabulary usually adopted by developers in Javadoc comments and, more in general, about this specific task. Second, the Javadoc has, by its nature, a more regular structure making use of tags (*e.g.,* `@param`) that could help the model in better predicting the comment, especially given the fact that the T5 model exploits the relevant code context during the prediction. Still, even on the $Inner\text{-}comment_{task}$ the T5 model achieves three times more perfect predictions of the 5-gram when predicting up to seven tokens (bottom-left corner of Fig. 9.3).

When the number of tokens to predict increases, the gap in performance between the two approaches gets thinner. In the most complex scenario in which the models predict more than 10 tokens in the comment, the T5 achieves, overall, 13% of perfect predictions against the 3% of the 5-gram model.

The difference is smaller for the $Inner\text{-}comment_{task}$, with the best approach (T5) achieving only 1% of perfect predictions.

The McNemar's test always indicates significant differences in terms of perfect predictions ensured by the T5 and the 5-gram model, with ORs ranging between 8.04 for the $Inner\text{-}comment_{task}$ and 17.56 for the $Javadoc_{task}$ (OR=16.79 for the overall dataset). The better performance of T5 is confirmed by the other evaluation metrics we adopted, namely the BELU-A and the Levenshtein distance. The BLEU-A gap is up to five times in favor of the T5 over the 5-gram, confirming a substantial difference in performance between the two models. On the $Javadoc_{task}$, T5 always achieves a BLEU-A score higher than at least ∼16% as compared to that achieved by the 5-gram, independently of the number of tokens the two models are asked to predict. As already observed for the perfect predictions, the BLEU-A gap is much smaller in the $Inner\text{-}comment_{task}$, however still showing a plus ∼5% in favor of T5 up to six tokens. The difference in performance tends to decrease while increasing the number of tokens to predict.

By focusing on the Levenshtein distance (right part of Fig. 9.3, the lower the better), we can observe that, as expected, the number of token-level edits needed to convert a prediction into the reference one tends to increase for both models when more tokens are predicted. An analysis of both tasks (*i.e.,* $Javadoc_{task}$ and $Inner\text{-}comment_{task}$) points out that T5 requires a developer intervention in a lower number of cases than 5-gram. However, a clear conclusion can be drawn by looking at the three graphs in the right part of Fig. 9.3: When the two models are not able to generate a perfect prediction, the effort required by developers to convert the prediction into the comment they actually want to write might be too high. For example, when predicting the next five tokens the developer is likely to type, the T5 requires, on average, to changes to 3.2 of the predicted tokens.

Task (d)	$Shared_t$	$OnlyT5_t$	$OnlyBL_t$
$Javadoc_{task}$	17.06%	75.87%	7.07%
$Inner\text{-}comment_{task}$	19.70%	67.78%	12.52%

Table 9.5. Perfect predictions overlap between T5 and *5*-gram

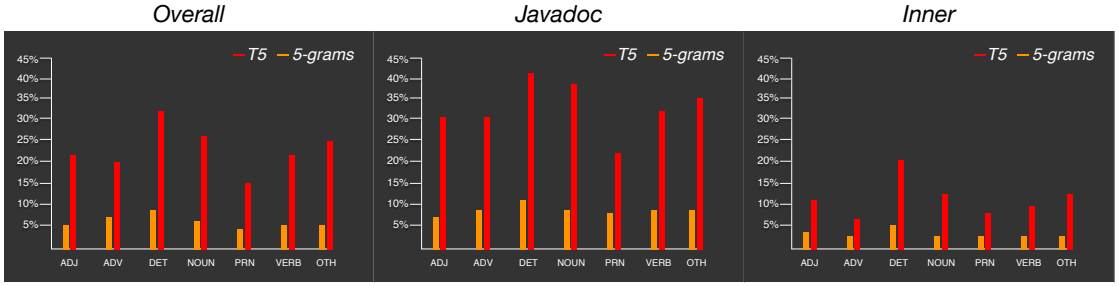


Figure 9.4. Part-of-speech tag analysis: ADJ=adjective, ADV=adverb, DET=determiner, PRN=pronoun, OTH=others which includes *conjunction, number, particle, adposition, and X*, where X are words that cannot be assigned a part-of-speech category

An important point to discuss in the comparison between T5 and 5-gram is the different datasets used for their training. Indeed, T5 benefited of a pre-training phase in which it exploited additional code that was not made available to the 5-gram during training. Thus, we performed an ablation study on the T5 model by removing its pre-training step and checking to what extent its superior performances are due to the performed pre-training. While details can be found in our replication package [repe], we can summarize our findings as follows: The pre-training phase increases the performance of the T5 in terms of perfect predictions in a range going from ∼0.5% to ∼2%, depending on the task and on the k value. Thus, while pre-training is beneficial, the performance of the T5 are still better than those of the 5-gram model even when both models are only trained on the fine-tuning dataset.

Table 9.5 reports the results of the overlap metrics we computed (see Section 9.2.2). For the $Javadoc_{task}$, only 17.06% of the perfect (*i.e.,* correct) predictions are shared among the two models, while 75.87% are correctly generated only by the T5 model. The *5*-gram is responsible for the remaining 7.07% of perfect predictions, that are missed by the T5. This shows, at least for the $Javadoc_{task}$, a limited (but existing) complementarity between the models. Similar results are achieved for the $Inner\text{-}comment_{task}$. The two models share 19.70% of perfect predictions, with 67.78% of them correctly predicted by T5 only. The *5*-gram model contributes the remaining 12.52% again showing some complementarity between the models.

Fig. 9.4 shows the POS analysis (*i.e.,* percentage of correct predictions of each POS type) that confirms the superior performance of T5 across all investigated POS categories: Independently from the type of word to predict, the T5 outperforms the *5*-gram model. Also, the performances on the $Javadoc_{task}$ are, as expected, superior. Not surprisingly, determiners are the ones having the highest percentage of correct predictions. Nevertheless, POS types like nouns, adjective, and verbs which are certainly more challenging to predict still

exhibit a good percentage of correct predictions. This analysis, combined with the previous one showing the performance of the model at different values of k, shows that the perfect predictions obtained by the two models (and in particular by the T5) are not only the result of trivial single word ($k = 1$) predictions involving simple POS types, but also include more challenging prediction scenarios. Fig. 9.5 reports five qualitative examples of predictions performed by both models: the first two are successful predictions made only by the T5 for the $Javadoc_{task}$ and the $Inner\text{-}comment_{task}$, respectively.

Note that for the first one, the 5-gram does not generate any prediction likely due to the fact that the 4-gram provided as input was never encountered in the training set. The T5, instead, correctly guesses the subsequent 12 tokens in the comment out of the 14 we masked (thus, this is a perfect prediction when considering $k = 12$). The third qualitative example is a prediction failed by both techniques, with the comment generated by the T5 model being more similar to the reference one. Finally, the two predictions at the bottom represent cases in which the 5-gram model correctly completes the comment while the T5 comment fails. Also in this case the first of the two examples is taken from the $Javadoc_{task}$, while the second represents an $Inner\text{-}comment_{task}$. The complete list of predictions is publicly available [repe].

Overall, our quantitative analysis showed the superiority of the T5 model in the code comment completion task. However, it is important to mention that the T5 model exploits during the prediction the *context* we provide as input (*i.e.,* the code likely to be relevant for the specific comment). This means that, from a practical point of view, such a model can be exploited for code comment completion only assuming that the developer first writes the code and, then, the comment to document it. Clearly, this is not always the case and limits the applicability of the T5 model we experimented with. Such a problem is instead not present in the n-gram model, that can always be applied as long as $n-1$ tokens are present before triggering the prediction of the n^{th} token. A tool integrated into an IDE to support code comment completion could exploit both models: The n-gram model could be triggered if no context can be captured for the comment being written, while the T5 can perform the prediction when relevant code is present.

Fig. 9.6 depicts the average *confidence* level for perfect (continues lines) and wrong (dashed) predictions made by the T5 (red lines) and the *5*-gram (orange) model. As it can be seen, the confidence of both models is a good proxy for the quality of the predictions. This is particularly true for the T5, for which we can see as the correct predictions tend to have a confidence greater than 0.5 for all k values, while the wrong predictions have confidence approaching 0.0 when k increases. Thus, a threshold based on confidence could be used in IDE tools to avoid recommending predictions likely to be wrong.

Finally, it is worth mentioning that, as compared to the task of code comment generation (*i.e.,* generating a comment from scratch given a code as input), the performance achieved in terms of perfect prediction is substantially higher. Indeed, when looking at a similar model (T5) experimented in the context of Javadoc generation, it achieved ∼10% perfect predictions [MSC+21]. The results in Fig. 9.3 show that, depending on the length of the prediction on the $Javadoc_{task}$, for code comment completion, the same model can achieve ∼16-55% perfect predictions. Thus, it might be more feasible for such a (simpler) problem to develop tools that can already support developers in their everyday coding activities.

Qualitative examples

```
Attempts to normalize a vector. If this fails, the <MASK>
public static Vector4f normalizeSafe(Vector4f v){
       try{
         return v.normalize();
       } catch(ArithmeticException ex){
         return Vector4f.ZERO;
       }
}

MASK:   method catches the exception and return a zero vector of the same
        dimension instead
T5:     method catches the exception and return a zero vector of the same
```

```
protected void onSizeChanged(int w, int h, int oldw, int oldh) {
       super.onSizeChanged(w, h, oldw, oldh);

       // Make sure <MASK>
       if (h != old) {
          recomputeScrollPosition(h, oldh, mPageMargin, mPageMargin);
       }
}

MASK:   scroll position is set correctly
T5:     scroll position is set correctly
5-Gram: there is capacity for at
```

```
public void write(ChannelHandlerContext ctx, Object message,
                  ChannelPromise promise) {
       throws Exception {
          checkState(
             state == State.PROTECTED,
             "Cannot write … %s", state);
       }
       …
       // Just <MASK>
       pendingUnprotectedWrites.add(msg, promise);
}

MASK:   add the message to the pending queue.
T5:     add the first one and try again.
5-Gram: update the new credentials.
```

```
@return An account limit structure that <MASK>
public java.util.List<AccountLimit> getAccountLimits(){

       if (accountLimits == null) {
           accountLimits =
              new com.amazonaws.internal.SdkInternalList<AccountLimit>();
        }
        return accountLimits;
}

MASK:   contain a list of AWS CloudFormation account limits and their
        values.
5-Gram: contain a list of AWS CloudFormation account limits and their
        values.
T5:     describes the layer's limits.
```

```
public  ByteBuffer encode(OffHeapValueHolder<V> valueHolder);
        ByteBuffer serialized;
        …
        byteBuffer.putLong(valueHolder.expirationTime());
        //represent the hits on previous versions. It is kept for
        //compatibility reasons with <MASK>
        byteBuffer.putLong(0L);
        …
        return byteBuffer;
}

MASK:   previously saved data
5-Gram: previously saved data
T5:     write/write
```

Figure 9.5. Qualitative examples.

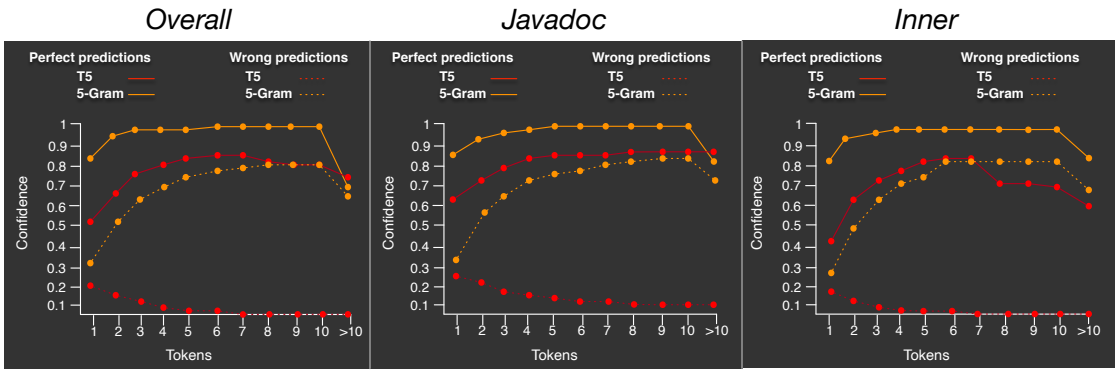


Figure 9.6. Confidence level in relation to the length of the predicted tokens

9.4 Threats to Validity

Construct validity. While building our dataset, we made three important assumptions. First, by providing the code context as input to the T5, we assume that the comment is written by developers when the code is already implemented, which is not always the case. Still, this does not invalidate the reported results, but it means, from a practical point of view, that comment completion recommendations could be triggered by the T5 only when comments are added after the code is written. Second, when fine-tuning the T5 model, we hid from the comment under analysis the sentences following the ones in which we masked tokens. Thus, we are assuming that the comment is written linearly (*i.e.,* one sentence after the other) which, again, is not always the case. Third, in our study we use the original comment written by developers as oracle, assuming that it represents a valuable reference for the experimented model. Also this assumption, while done in many previous works [HLX+18a, IKCZ16, APS16b, HLX+20b, HLWM20], might be wrong.

Internal validity. An important factor that influences DL performance is hyperparameters tuning. For the pre-training phase, we used the default T5 parameters selected in the original paper [RSR+20]. For the fine-tuning, we did not change the model architecture (*e.g.,* number of layers), but we experimented with different learning rates. We are aware that a more extensive calibration would likely produce better results.

External validity. Our study involved ∼500k Java methods, thus ensuring a good generalizability at least for Java code. Our results cannot be generalized to other languages.

9.5 Conclusions

We presented an empirical study comparing two different techniques, namely the T5 and the *n*-gram model, in the task of code comment completion (*i.e.,* autocomplete a code comment the developer started writing). The two models are different in nature, with the T5, based on deep learning, exploiting as information to support the completion of a comment C not only the first tokens typed by the developer while writing C but also a "context" representing code

relevant for C. The n-gram model, instead, does only consider the $n-1$ preceding tokens to predict the n^{th} token.

Our results showed the superiority of the T5, achieving significantly better prediction performance as compared to the n-gram model. However, the simplicity of the latter and its wider applicability (it does not require a code context as input), make the two models potentially complementary in the implementation of a code comment completion tool.

10

A New Metric for Evaluating Code Summarization Techniques

Empirically evaluating the quality of code summaries generated by automated approaches is far from trivial. Indeed, assessing the extent to which a natural language text represents a good summary for a code component would require human (developers) judgment. Given the difficulties of running large-scale evaluations with developers, the software engineering community borrowed evaluation metrics from the Natural Language Processing (NLP) field. These include (but are not limited to) BLEU [PRWZ02], ROUGE [Lin04], and METEOR [BL05]. These metrics have been originally designed to act as a proxy for the quality of automatically generated text (*e.g.,* a translation) by comparing it with a reference (expected) text: The higher the words' overlap between the generated and the reference text, the higher the assessed quality.

When adopting such metrics for code summarization, the generated summary is contrasted against a single reference text, usually being the original comment written by developers for the code provided as input, which can be easily mined from software repositories. However, two important issues emerge when using the NLP metrics in this context. First, there is no guarantee that the reference text is of high quality, as also demonstrated by empirical studies documenting quality issues in code comments (see *e.g.,* [FWG07, LVLVP15, WNBL]). Thus, computing a word overlap between the generated and the reference summary may provide misleading indications of the actual quality of the generated summary. Second, the mined comment is only one of the possible ways to summarize the related code. Metrics based on word overlap penalize generated summaries for being different but semantically equivalent to the reference one, thus again not being good proxies for the summary quality.

The software engineering research community is well aware of the limitations of these metrics [RFA21, HEBM22]. In response to the second limitation previously discussed (*i.e.,* capturing similarity between summaries using different wording but being semantically equivalent), Haque *et al.* [HEBM22] proposed the usage of word/sentence embeddings to properly capture the semantic similarity across summaries, moving from word overlapping to word-similarity measurement. While the authors show that these metrics better correlate

with the human judgment of code summary quality as compared to word-overlapping metrics (*e.g.,* BLEU, ROUGE, METEOR), they still suffer of the first limitation we discussed, *i.e., a high similarity to a low-quality reference summary may provide a misleading good assessment of a generated summary*.

In this research, we argue that an important factor in the assessment of summary quality is currently ignored by the state-of-the-art metrics: "The suitability of the generated summary for the code to document, independently from the original comment written by developers". To provide evidence of that, we present an empirical study that analyzes to what extent, different metrics to assess automatically-generated code summaries correlate, and complement with each other in explaining human assessment of such summaries. More importantly, we check for the complementarity of metrics capturing the new dimension with respect to others.

We rely on a dataset by Roy *et al.* [RFA21], featuring more than 5k human evaluations of automatically generated summaries. Particularly, the authors focus on the interpretation of metrics used in code summarization, and in particular on BLEU, ROUGE (in several different variants), METEOR, chrF [Pop15], and BERTScore [ZKW+19]. They conducted a study with researchers and practitioners asking them to assess the quality of 36 summaries associated with 6 code snippets (a Java method). Each snippet had six summaries associated, one being the reference summary (*i.e.,* the one written by the original developers) and five resulting from different code summarization techniques [LJM19, HLWM20, ABLY18, XWW+18, VSP+17]. Overall, they collected 226 surveys, for a total of 6,253 evaluations (not all participants fully completed the survey. The quality assessment has been performed using an ordinal scale in the range [0, 5] [Opp92] (the higher the better) and focusing on three different aspects of each summary: conciseness, fluency, and content adequacy. We relate such evaluations to the quality assessment provided by different families of metrics.

To capture the suitability of the generated summary for the code to document we experiment with (i) a simple approach relying on the word overlap between the summary and the code [SHJ13]; (ii) a deep learning-based approach exploiting embeddings obtained via a model pre-trained on code [WLG+23]; and (iii) `SIDE` (**S**ummary al**I**gnment to co**D**e s**E**mantics), a new metric leveraging contrastive learning [SKP15] to model the characteristics of suitable and unsuitable code summaries for a given code. Our results show that (i) focusing on assessing the suitability of the generated summary for the documented code ignoring the reference summary allows to capture orthogonal aspects of summary quality as compared to state-of-the-art metrics, such as BLEU, ROUGE, METEOR, Jaccard similarity, as well as metrics based on word/sentence embeddings; and (ii) `SIDE` is the metric having the strongest correlation with human judgment of code summary quality. We also show that `SIDE` can be combined with state-of-the-art metrics to provide a more comprehensive assessment of code summary quality.

10.1 SIDE

We present `SIDE`, our novel metric to assess whether a natural language text represents a suitable summary for a given code. First, we provide background information about the DL

model on top of which SIDE is based (Section 10.1.1), and the contrastive learning procedure used to train it (Section 10.1.2). Then, we describe the dataset used for the model's training (Section 10.1.3). Finally, Section 10.1.4 provides the details (*e.g.,* parameters) of the training procedure.

10.1.1 MPNet in a Nutshell

MPNet (Masked and Permuted Pre-training for Language Understanding) [STQ+20] is a Transformer [VSP+17] pre-trained model built on top of BERT [DCLT19]. Before discussing its architecture, let us briefly introduce the notion of pre-trained models. Pre-trained Transformers achieved state-of-the-art results in several Natural Language Processing (NLP) tasks [DCLT19, YDY+19, LOG+19, RSR+20, GRL+21, YDY+19, STQ+20, LLG+20, ACRC21, ZZSL20b]. The pre-training, together with the self-attention mechanism featured in the Transformer architecture [VSP+17], played a major role in these achievements. The idea of pre-training is to provide the model with general knowledge about a language of interest before specializing it for a specific task. For example, let us assume we want to create an English-to-French translator. The model can be pre-trained on a large amount of unlabelled English and French data (*e.g.,* articles extracted from Wikipedia) using a self-supervised training objective, such as masked language modeling (MLM). MLM consists in randomly masking a percentage of the tokens in a given (English or French) sentence asking the model to guess them. For example, applying MLM to a sentence $\langle T_1, T_2, T_3, T_4, T_5 \rangle$ we could obtain $\langle T_1, M, T_3, M, T_5 \rangle$ (*i.e.,* T_2 and T_4 have been masked). The input for the model is the masked sentence, while the expected output are T_2 and T_4 as replacements for the two masked tokens. The idea is that thanks to MLM, the model starts acquiring knowledge about the languages' structure, preparing it to be specialized (fine-tuned) for the task of interest, namely language translation.

The MLM pre-training objective has been adopted by several Transformer architectures, such as BERT [DCLT19]. However, MLM suffers from a limitation: It ignores the dependencies among the masked tokens, possibly limiting the learning of complex semantic relationships. To overcome this limitation, Yang *et al.* [YDY+19] proposed in XLNet the usage of permuted language modeling (PLM) during pre-training. This forces the model to learn long-range relations between tokens by guessing the correct positioning of the tokens in the whole sentence. While the technical details can be found in the paper introducing the technique [STQ+20], the basic idea is that at pre-training time the model is provided with a permuted sentence featuring masked tokens and, on top of that, with original positioning information, *i.e.,* what was the original position of the tokens in the sentence before permutation. MPNet managed to achieve new state-of-the-art results in several works from the NLP community [PHSP22, MHH+22, GDX+21, HTZ+21], including those relying on contrastive learning as training procedure [PHSP22, WCH+22], and it is the pre-trained model we specialize for the task of classifying a textual summary as suitable or not for a given code.

In terms of architecture, MPNet builds upon the $BERT_{base}$ model, which comprises 12 transformer layers with a hidden size of 768, 12 attention heads, and a total of 110M trainable parameters. MPNet was pre-trained using the same corpora exploited for the training of

RoBERTa [LOG+19], which includes datasets such as Wikipedia and BooksCorpus [ZKZ+15], OpenWebText [GC19], CC-News [ccn], and Stories [ccs], summing up for a total of 160GB of textual data.

10.1.2 Contrastive Learning

Contrastive learning [SKP15] allows DL models to learn an embedding space where similar sample pairs (*i.e.,* pairs sharing specific features) are clustered together while dissimilar pairs are set apart. In our context, we use contrastive learning to discriminate suitable *vs* unsuitable summaries for a given source code snippet. To this aim, we need to show to the model both *positive samples* (source code associated with suitable summaries) and *negative samples* (source code associated with unsuitable summaries).

Several contrastive representation learning losses have been proposed in the literature [HCL06, SKP15, Hub92, Kul97]. We employ the *triplet loss* [SKP15], which has been shown to better encode the positive/negative samples as compared to other contrastive losses [CHL05]. The triplet loss function has been proposed by Schroff *et al.* [SKP15] and introduces the concept of "anchor". Given an anchor x, a positive (x^+) and a negative (x^-) sample is selected, with the triplet loss which during training minimizes the distance between the x and x^+, while maximizing the distance between x and x^-.

In our case, the anchor is the code to document, with a suitable summary representing x^+ and an unsuitable summary representing x^-. In the following, we introduce the dataset used to fine-tune MPNet for the task of interest, explaining how we generate positive and negative samples.

10.1.3 Fine-tuning Dataset

We need to collect code instances paired with "suitable" and "unsuitable" code summaries. In our evaluation of SIDE (Section 10.2) we will exploit a dataset from the literature featuring developers' evaluation of code summaries for Java methods [RFA21]. Thus, we started by collecting Java methods accompanied by a textual summary.

We exploited the CodeSearchNet dataset [HWG+19], featuring ∼6M functions from various programming languages, including Java, and a subset of Java methods is accompanied by a Javadoc description. Lu *et al.* [LGR+21] observed that the original CodeSearchNet dataset featured instances potentially being problematic. Therefore, they created a curated version of the dataset excluding methods that cannot be parsed, and those paired with a Javadoc (usually, the method's summary) having its first sentence shorter than 3 or longer than 256 tokens, and featuring special tokens (such as $\langle img \rangle$), or not being written in English. Their refined Java subsection of the CodeSearchNet dataset comprises 181,061 pairs of $\langle method, summary \rangle$ which have been already split into three subsets: 164,923 training, 5,183 validation, and 10,955 testing. The $summary$ here is the first sentence of the Javadoc extracted as the first paragraph of the documentation (*i.e.,* the one delimited with the first period) [LGR+21]. To increase the confidence in the quality of the exploited dataset and verify whether the sentences automatically extracted from the documentation actually represent summaries of the method, two of the authors independently inspected 100 randomly

selected samples from the dataset, classifying their associated documentation as "code summary" or "other". The guideline was to classify it as a code summary if it summarizes the intent of the method. After solving 2 cases of disagreement, 95 of the inspected documentations were classified as actual summaries.

The starting assumption is that the original summary written by the developers is a positive sample when paired with its associated method. This makes for 164,923 positive samples in our dataset. The same number of negative samples can be easily created by associating each method with a randomly selected summary from the training set (different from the original summary). The result is a dataset of 164,923 $\langle method, positive, negavite \rangle$ triplets featuring, for each method, a positive and a negative summary. The approach used to create the negative samples, while simple, may associate unrelated summaries to methods, simplifying the learning of the model, *i.e.,* it becomes rather easy to discriminate between positive and negative samples at training time. However, when assessing the quality of an automatically generated summary (our final goal for `SIDE`), it is unlikely that the latter is completely unrelated to the input method, even when it is of low quality. Thus, we must train `SIDE` so that it is able to identify as "unsuitable" for a given method even summaries which are plausible yet still suboptimal. These are known in the literature as hard-negative samples [OSXJS16, SKP15].

To automatically generate these summaries, we conjecture that inner comments only documenting a subset of the method's statements are unsuitable as "method's summary". Nevertheless, they are still likely to be related to the method, certainly more than randomly selected summaries. Thus, we parsed the methods in the training set to extract all their inner comments and associated each inner comment to the set of statements it documents. For such association, we use the heuristics previously proposed in the literature, linking each inner comment to all following statements until an empty line or a closing curly bracket is reached [CHL$^+$19b, HHC$^+$20]. Comments reporting self-admitted technical debt [PS14a] have been identified using keywords matching and excluded (we removed all comments including one of the following words: *to-do, fix-me, todo, fixme, xxx, hackme, hack-me*).

Indeed, these comments do not describe the code. We then computed the percentage of statements in the method that each comment documents, only considering actual code statements (*i.e.,* excluding comments and blank lines). We consider a comment as a good hard-negative sample (*i.e.,* a plausible method description being, however, unsuitable as a method's summary) if it documents less than 25% of the method's statements. The choice of the 25% threshold is motivated by the following assumption: if a comment describes at most one fourth of the statements in a method, it is unlikely that it can represent a comprehensive summary of it while still possibly documenting some of the responsibilities implemented in the method. We managed to create a hard-negative for ∼15% (24,951) of the instances in our training dataset, since the others did not have any inner comment matching our selection procedure.

Our training dataset features 164,923 triplets generated using random negatives, and 61,001 triplets featuring instead hard-negatives, since each of the 24,951 methods for which we managed to create a hard-negative can contribute with more than a single instance.

10.1.4 Training and Model Evaluation

MPNet has been trained for 10 epochs (which accounted for 141,205 training steps) using a batch size of 16 and a maximum sequence length of 512 tokens. The sequence length acts on the input of the model (in our case, the concatenation of a method and its summary), cutting out longer sequences. This impacted only 4.27% of the instances in our fine-tuning dataset. The learning rate has been warmed-up by taking into account the overall size of the training dataset, batch size and number of epochs. This strategy, which increases the learning rate from 0 to 2e-5 (default values when using the AdamW optimizer [LH19]) has been previously adopted when training BERT-based models using Sentence Transformers [RG19]

The best-performing checkpoint which we found to be the last one saved (*i.e.,* after 141,205 training steps) has then been selected as the one maximizing the following score (also used in previous work using contrastive learning [ZKW+20, SO21]):

To reduce the chance of overfitting, we save checkpoints every 5k training steps while using a patience of 5. Specifically, we evaluate the model on the validation set (Section 10.1.3), which includes 5,183 positive samples (the original descriptions associated with each method) and the same number of negative samples randomly generated as explained in Section 10.1.3. The best-performing checkpoint has then been selected as the one maximizing the following score (also used in previous work using contrastive learning [ZKW+20, SO21]):

$$\frac{\sum_{i=1}^{N} \text{CSpositive}_i - \text{CSnegative}_i}{N}$$

where N represents the number of positive and negative samples (which in the evaluation set is guaranteed by construction to be the same), CSpositive_i represents the cosine similarity returned by the model between the method m_i and its "positive" summary, while CSnegative_i represents the cosine similarity returned between method m_i and its "negative" summary. A perfect model would return 1.0 for such a metric, reporting 1.0 as the similarity of all positive summaries and -1.0 as the similarity of all negative summaries. We acknowledge that our evaluation set features randomly generated negative instances that, as such, are simple to identify by the model.

However, such a procedure is just used to select the best-performing checkpoint for `SIDE`, and not for its empirical evaluation (described in Section 10.2).

10.2 Study Design

The *goal* of this study is to evaluate metrics used in the literature to assess the quality of code summarization approaches, including our newly proposed metric `SIDE` using contrastive learning. The *quality focus* is the complementarity of the considered metrics with respect to others, and the extent to which the considered metrics would explain summary quality assessments performed by developers. The *perspective* is of researchers wanting to define a framework for the assessment of code summary quality, that can be used, for example, to evaluate code (re)documentation approaches.

The *context* consists of (i) the dataset by Roy *et al.* [RFA21] featuring more than 5k developers' quality assessments of automatically generated summaries; and (ii) 40 summary evaluation metrics, including SIDE. The study addresses the following research questions:

RQ$_1$: *To what extent different metrics to evaluate the quality of source code summarization correlate with each other?* Before analyzing how metrics contribute to explaining the quality of generated summaries, we analyze the extent to which they are related with each other, or, instead, capture different dimensions of the dataset variability.

RQ$_2$: *To what extent different metrics to evaluate the quality of source code summarization contribute to explain user-based evaluations?* After having identified a subset of unrelated metrics capturing the dataset variability, we analyze the extent to which these metrics contribute to explaining different summarization quality indicators. This would aim at providing a quantitative indication not only of the complementarity of the different metrics but also of the importance of each metric to describe an extrinsic quality indicator.

10.2.1 Evaluation Dataset

We use the dataset by Roy *et al.* [RFA21]. As previously explained, this dataset features humans' evaluations of both automatically generated summaries for Java methods as well as of the original summaries written by the methods' developers. The latter are not suitable for our study. Indeed, to compute some of the evaluation metrics (*e.g.,* BLEU score), we need an automatically-generated summary to contrast against a reference summary. Thus, we remove from the set of 6,253 evaluations the 1,052 evaluations referring to the (manually written) original summaries, leaving us with the dataset of 5,201 evaluations.

10.2.2 Variable Selection

The 5,201 evaluations performed by developers [RFA21] concern three quality aspects of summaries, all rated on an ordinal scale from 0 to 5 [Opp92] (the higher the better):

- **Conciseness:** Assesses the degree to which the summary contains unnecessary information.
- **Fluency:** Evaluates the continuity or smoothness rate in the generated summary.
- **Content Adequacy:** Assesses the extent to which the summary lacks information needed to understand the code.

In addition Roy *et al.* [RFA21] also collected a Direct Assessment (DA) score [GBMZ13], expressed on a scale from 0 to 100, and providing an overall quality assessment of the summary. The three quality aspects (conciseness, fluency, and content adequacy), and the DA score represent the *dependent variables* of our study.

Regarding the *independent variables i.e.,* automated evaluation metrics for code summaries we considered, besides SIDE, (i) all metrics in the work by Roy *et al.* [RFA21], (ii) additional metrics used in the work by Haque *et al.* [HEBM22], (iii) the c_coeff metric proposed by Steidl *et al.* [SHJ13], and (iv) a baseline we define based on CodeT5+ [WLG$^+$23]

(a Transformer pre-trained on code and English text) to compute the textual similarity between the generated summary and the code. The latter has the purpose to let us check the actual benefits (if any) brought by the contrastive learning we adopt in SIDE as compared to other DL-based approaches. The considered metrics, besides being a comprehensive set of those used in the literature to assess code summarization techniques, approach the quality assessment of code summaries in different ways.

In particular, the ones inherited from the works by Roy *et al.* [RFA21] and Haque *et al.* [HEBM22] are conventional metrics, in the sense that they look at the similarity between the generated summary and the reference summary. These metrics range from very simple (*e.g.,* word overlapping between the two summaries), to more complex ones exploiting DL-based embeddings. Differently, c_coeff [SHJ13], CodeT5+, and SIDE look for the similarity between the generated summary and the documented code. This is the first time such a dimension is considered in the assessment of code summarization techniques. While several approaches can be used to measure such a similarity, we opted for three metrics including a "trivial" solution (*i.e.,* the c_coeff metric exploiting word overlap), a state-of-the-art Transformer pre-trained model (CodeT5+), and a contrastive learning-based solution being SIDE. For what concerns SIDE, we consider two of its variants: One exploiting hard-negatives in the training set (as extracted using the procedure described in Section Section 10.1.3) and one only including random negatives. This is basically an ablation study investigating the role played by the hard-negatives on the ability of SIDE to assess the suitability of a summary for a given code.

10.2.3 Words/characters-overlap based Metrics

BLEU (BilinguaL Evaluation Understudy) [PRWZ02] measures the similarity between the candidate (predicted) and reference (oracle) summaries. Such a similarity assesses the overlap in terms between the two summaries and it is defined on a scale between 0 (completely different summaries) to 1 (identical summaries). We compute the BLEU score at the sentence level for various values of n, including n={1, 2, 3, 4}. We also compute the BLEU-A being the geometric mean of the four BLEU variants we consider.

METEOR (Metric for Evaluation of Translation with Explicit ORdering) [BL05] is computed as the harmonic mean of unigram precision and recall, with recall being assigned a higher weight than precision. In contrast to BLEU, METEOR incorporates stemming and synonyms matching to align more closely with human perception of similarity between sentences. The METEOR score ranges from 0 to 1, with a score of 1 indicating two identical sentences.

ROUGE (Recall-Oriented Understudy for Gisting Evaluation) [Lin04] is a set of metrics for evaluating both automatic summarization of texts and machine translation techniques. ROUGE metrics compare an automatically generated summary or translation with a set of reference summaries (typically, human-produced). Similarly to Roy *et al.* [RFA21], we compute ROUGE-N(1-4), ROUGE-L, and ROUGE-W. ROUGE-N measures the number of matching n-grams between the generated summary and the reference summary with results reported in terms of recall, precision and f1-score.

Jaccard [Han04] measures the degree of overlap between two sets of tokens (summaries in our case). It is calculated by dividing the size of their intersection by the size of their union, thus obtaining a value between 0 and 1 (the higher the Jaccard, the more similar the two summaries).

chrF (character *n*-gram F-score) [Pop15] measures the similarity between the generated and the reference summaries at the character level (rather than at the word-level as done by the above metrics), reporting the computed value using the F-score.

c_coeff [SHJ13], while still based on word overlap, focuses on the similarity between a summary and its associated code: It computes the percentage of words in a summary that is similar to words in the code, where two words are considered similar if they have a Levenshtein distance lower than two.

10.2.4 Embedding-based Metrics

TF-IDF (Term frequency-inverse document frequency) [R+03] is a widely used term weighting schema assessing the importance of a word within a document collection. In our context, TF-IDF is used to compute the cosine similarity (TF-IDF_CS) and the Euclidean distance (TF-IDF_ED) between the terms' vectors representing the generated and the reference summary.

BERTScore [ZKW+19] computes sentence similarity using the embedding of the BERT model [DCLT19], which has been trained on English textual data. We report all the BERTScores which include, precision (BERTScore-P), recall (BERTScore-R), and F1-score (BERTScore-F1).

SentenceBERT [RG19] employs a siamese network architecture to generate fixed-length representations of sentences using BERT [DCLT19] as a backbone to produce the encoding. The representations of the generated and the reference summary are compared via cosine similarity (SentenceBERT_CS) and the Euclidean distance (SentenceBERT_ED).

InferSent [CKS+17] relies on GloVe vectors [PSM14] as pre-trained word embeddings for the sentence pair. The embeddings are then passed through RNN encoder layers to obtain fixed-length vector representations for each sentence. Also in this case both the cosine similarity (InferSent_CS) and the Euclidean distance (InferSent_ED) are considered to contrast the generated and the reference summary.

Universal Sentence Encoder (USE) [CYK+18] employs transformer encoders to generate context-aware representations of words within a sentence by levering the self-attention mechanism. Both USE_CS (cosine similarity-based) and USE_ED (Euclidean distance-based) are considered.

CodeT5+_CS [WLG+23] exploits CodeT5+, a model pre-trained on code and natural language. We use the CodeT5+$_{base}$ variant (∼220M trainable parameters) to compute the cosine similarity between the embeddings of the generated summary and the code to document. CodeT5+_CS acts as a further baseline for `SIDE` which has been fully trained via contrastive learning.

10.2.5 Analysis Methodology

In the following, we describe the study analysis methodology. The whole analysis has been performed using the *R* [R C20] statistical environment. For statistical tests, we assume a significance level $\alpha = 0.05$.

To address RQ_1, we first analyze the correlation between different summary evaluation metrics. To avoid being constrained with linear relationships only, we leverage the non-parametric, Spearman's rank correlation [Con98]. To show the correlation, we create a visual overview of correlation among metrics using the *varclus* function part of the *R Hmisc* package [HwcfCDmo20]. The output of this procedure is a hierarchical clustering of variables (*i.e.,* our metrics), producing a dendogram (*i.e.,* the clustering tree) visualizing correlated metrics. Note that we do not use Spearman's correlation to select uncorrelated variables but, as explained below, the *redun* procedure which allow accounting for multicollinearity other than collinearity.

We complement the correlation analysis with a Principal Component Analysis (PCA) [Jol86]. PCA leverages Singular Value Decomposition (SVD) to describe the underlying data variance and covariance expressed as linear combinations of the considered variables.

To avoid having the results of the PCA being affected by collinearity, we run a variable selection procedure using the *redun* function from the *Hmisc* package [HwcfCDmo20]. This function performs a stepwise removal of the independent variables determining how well each variable can be predicted by the remaining ones. The process starts by first removing the most predictable variable and continuing subsequently until no variable among the predictors can be predicted with a given (adjusted) R^2, which we set equal to 0.8. Then, before applying PCA, we re-scale the variables in the range [0,1] using a min-max re-scaling.

The PCA on the selected variables returns the eigenvectors related to each independent component, where the (absolute) values in correspondence of each metric indicate the importance of the metric for that component. Moreover, it returns the standard deviation captured by each principal component, which indicates the importance of the component itself. The PCA has been performed using the *prcomp* function of the *R* (default) *stats* package.

To address RQ_2, we determine the extent to which different summary quality metrics can complement each other to explain extrinsic quality indicators provided by users. Given the (ordinal) scale of the dependent variables, and given that we cannot assume a linear relation between independent and dependent variables, this analysis has been performed by employing a multivariate logistic regression. A conventional logistic regression model is not suitable for our analysis, because the dependent variables are not dichotomous but, rather, expressed on a 0-5 ordinal scale [Opp92], or on a 1-100 scale (in the case of DA score).Therefore, we use an ordered logistic regression. Given the l levels of an ordinal variable, the ordinal logistic regression models ℓ different logits as:

$$ln\left(\frac{P(Y \leq \ell)}{P(Y > \ell)}\right) = \xi_\ell - \eta_1 X_1 - \cdots - \eta_n X_n \tag{10.1}$$

where Y is the model's dependent variable, X_i the independent variables, η_i their coefficients (estimates), and ξ_ℓ the intercept.

We use the *polr* function from the *MASS* package [VR02]. The interpretation of the model estimates is similar to the ones of logistic regression, with the difference that, given an estimate η_i, the OR=e^{η_i} indicates what are the increased odds of a unitary value increment for the dependent variable given a unitary increase of an independent variable. The application of the ordered logistic regression follows three subsequent steps, *i.e.,* (i) variable selection, (ii) variable re-scaling, and (iii) model building.

For the variable selection, we use the output of the *redun* procedure used in RQ_1. Then, given the selected variables, we re-scale them in the same range of the dependent variable being modeled. This makes the interpretation and comparison of the model's ORs easier. As for the dependent variables, we leave them as they are (*i.e.,* on a scale from 0 to 5 for *conciseness*, *fluency*, and *content adequacy*, and on a scale from 1-100 for *DA score*).

Finally, we apply the *polr* procedure, relating each dependent variable to all independent variables that were left after the *redun* feature selection. We report (i) fitting diagnostics (in particular the Akaike Information Criterion - AIC value), (ii) independent variables estimates, OR, and significance p-value adjusted, due to multiple factors, with the Benjamini-Hochberg correction [BH95].

10.3 Results Discussion

RQ_1: Correlation of different metrics to evaluate the quality of source code summarization

Fig. 10.1 depicts the hierarchical clustering of the independent variables output of the *varclus* procedure. Note that the values on the x-axis show the square of the Spearman's correlation (*e.g.,* a value of 0.64 indicates a correlation of $0.8^2 = 0.64$).

The top-level fork of the tree splits the metrics into two different families. The first groups all conventional metrics capturing the similarity between a generated and a reference summary, while the second includes the three metrics focusing on the relevance of the generated comment for the documented code (*i.e.,* `CodeT5-plus_CS`, `c_coeff`, and `SIDE`). This is a first indication of the fact that the two families of metrics focus on different aspects of summary quality.

Looking inside the sub-tree composed of conventional metrics, `BLEU-1` is not correlated with all others, not even with the other BLEU variants. While this may look counter-intuitive, `BLEU-1` focuses on single-word overlap, while its other variants look at n-grams overlap. This usually results in much higher values for the `BLEU-1` as compared to `BLEU-2`, `BLEU-3`, and `BLEU-4`.

The deeper we go into the tree, the more cohesive the clusters of metrics that can be observed. The three metrics looking at the documented code (*i.e.,* `CodeT5-plus_CS`, `c_coeff`, and `SIDE`) are very high in the dendogram indicating that, while they stay apart from the other conventional metrics, the correlation among them is small, likely due to the different underlying solutions used to assess the relevance of a generated summary for a given code.

The next step in our data analysis is to use the PCA to identify the orthogonal components in the performed measurements. To run it, we first selected the metrics to consider using

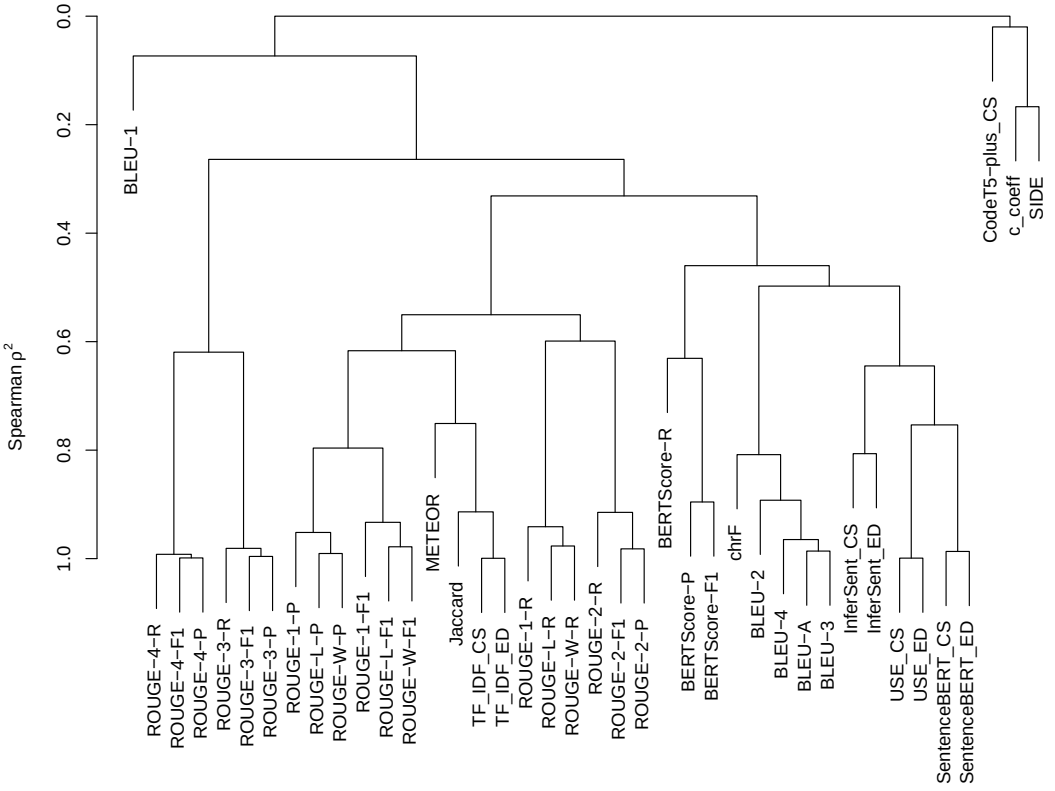


Figure 10.1. RQ_1 - Spearman's correlation clustering on the independent variables with varclus

the *redun* function as explained in Section 10.2.5.

This process resulted only in the 10 metrics listed in Table 10.1, already providing a first indication of the fact that the other metrics considered in our study are redundant and highly correlated with at least one of the 10 selected. The latter include (i) four words/characters-overlap based metrics capturing the similarity between the generated and the reference summary (*i.e.,* `BLEU-1`, `ROUGE-1-P`, `ROUGE-4-R`, and `ROUGE-W-R`); (ii) three embedding-based metrics also focusing on the similarity between the generated and the reference summary (*i.e.,* `BERTScore-R`, `SentenceBERT_CS`, and `InferSent_CS`); and (iii) all three metrics focusing on the similarity/relevance of the generated summary to the documented code (*i.e.,* `c_coeff`, `CodeT5-plus_CS`, and `SIDE`).

Table 10.1 reports the results of the PCA, with 10 principal components (PCs) identified. The first row indicates the proportion of variance captured by each PC. The higher such a value, the higher the variance in the dataset described by the PC. The second row reports the cumulative proportion of variance when considering the first *n* PCs. For example, by just using five PCs, it is possible to capture 91% of the variance in the data, *i.e.,* 55% (PC1) + 16% (PC2) + 8% (PC3) + 7% (PC4) + 4% (PC5). The remaining 10 rows in Table 10.1 show the importance of each metric for each PC: The higher the absolute value,

Table 10.1. RQ_1 - PCA for the evaluated metrics

	PC1	PC2	PC3	PC4	PC5	PC6	PC7	PC8	PC9	PC10
Prop. of Variance	0.55	0.16	0.08	0.07	0.04	0.03	0.02	0.02	0.01	0.01
Cumulative Prop.	0.55	0.71	0.80	0.87	0.91	0.94	0.96	0.98	0.99	1.00
BLEU-1	0.26	0.24	-0.44	0.11	-0.50	0.13	-0.44	0.45	-0.09	-0.09
BERTScore-R	0.47	0.15	-0.24	0.25	-0.18	-0.18	0.74	-0.07	0.10	-0.09
SentenceBERT_CS	0.38	-0.10	0.04	0.17	0.22	-0.68	-0.37	-0.18	-0.32	-0.18
InferSent_CS	0.23	-0.01	-0.01	0.06	0.09	-0.21	-0.17	0.10	0.54	0.75
ROUGE-1-P	0.44	0.07	-0.10	0.21	0.33	0.63	-0.17	-0.43	-0.14	0.11
ROUGE-4-R	0.41	0.26	0.38	-0.73	-0.25	0.01	-0.02	-0.14	-0.05	0.02
ROUGE-W-R	0.30	-0.04	0.36	0.06	0.44	0.15	0.05	0.64	0.19	-0.32
c_coeff	0.13	-0.47	0.53	0.39	-0.54	0.13	-0.07	-0.10	0.06	-0.01
CodeT5-plus_CS	-0.00	-0.02	-0.17	-0.08	-0.03	-0.02	-0.23	-0.34	0.72	-0.53
SIDE	0.20	-0.79	-0.39	-0.39	0.03	0.07	0.08	0.10	-0.07	0.02

the higher the metric's contribution to that PC. For each PC, we highlight with a dark background the metric(s) contributing the most to it. In particular, we highlight the metric with the highest absolute value in the eigenvector and all those close to it (at most -5% within the absolute value). The "5%" choice is arbitrary and just meant to simplify the results' visualization and discussion highlighting all metrics similarly contributing to a PC. PC1 is captured by conventional metrics looking for textual similarity between the generated and the reference summary (mostly `BERTScore` and `ROUGE`). PC2 is instead exclusively captured by `SIDE`, suggesting the importance of including metrics considering the code to document when measuring summary quality.

The second-highest value for PC2 is obtained by using `c_coeff`, but it is substantially smaller than the one associated with `SIDE` (0.47 *vs* 0.79). For PC3, `c_coeff` exhibits instead the highest eigenvector.

The analysis conducted so far indicates that by just considering three metrics (*e.g.,* BERTScore-R, SIDE, and c_coeff) it is possible to capture 80% of the variance in the data.

Conventional metrics are also associated with PC4 (`ROUGE-4-R`), with `c_coeff` being instead the one capturing PC5. Basically, among the top-5 PCs, 3 are mostly captured by metrics considering the code to document in the equation.

In summary, RQ_1 findings show that conventional metrics looking at the similarity between the generated and the reference summary do not correlate with those looking at the relevance/similarity of the generated summary for/with the documented code. Also, they only capture part of the variance in the data.

This only indicates that the metrics considering the documented code (*e.g.,* `SIDE`, or `c_coeff`) capture orthogonal information as compared to the conventional ones, not whether or not they better capture code summary quality as perceived by developers.

RQ_2: Contribution of different code summarization metrics in explaining user-based evaluations

Before discussing how various metrics to assess source code summarization impact user-based evaluations, we must anticipate that, in this section will focus exclusively on the outcomes achieved when training SIDE using hard-negatives. Additional information about the conducted ablation study is provided in Section 10.3.2.

Table 10.2 reports the results of the ordered logistic regression for each code summary quality attribute (dependent variables) manually evaluated by developers (*i.e., DA score*, *content adequacy*, *conciseness*, and *fluency*). We focus our discussion on the Odds Ratios (ORs) of the statistically significant *p*-values (<0.05), which are the ones highlighted with a black background. The interpretation of the ORs is as follows: they indicate the odds of a unitary increment in the dependent variable given a unitary increment in the independent variable. For example, if we look at the OR of SIDE when considering the *content adequacy* as a dependent variable (1.6265), it indicates ∼62% higher odds of observing a unitary increment in the *content adequacy* as perceived by developers for each unitary increment of the SIDE value. Remember that both independent and dependent variables have been normalized on the same scale: In the case of *DA score*, all variables are in the 1-100 range, while the three other variables are in the range 0-5, since we followed the scales used in the developers' evaluation. This also explains the lower ORs in the *DA score* table. Indeed, a unitary increase on a 1-100 range has a different magnitude as compared to a unitary increase on a 0-5 scale.

A first observation that can be made is that SIDE is the metric having the highest OR independently from the considered dependent variable. Also, only five metrics obtained a significant *p*-value for at least one dependent variable (SIDE is always among them).

The "overall" quality of the summary (*i.e., DA score*) and its *content adequacy* as perceived by developers is captured by five metrics: SIDE, c_coeff, CodeT5-plus_CS, SentenceBERT_CS, and ROUGE-1-P, with SIDE and c_coeff having the highest and second-highest OR, respectively, for both dependent variables. Still, conventional metrics looking at the similarity between a generated and a reference summary also play a role in capturing code summary quality as assessed through *DA score* and *content adequacy*.

When moving to the summary *conciseness* and *fluency*, SIDE and c_coeff confirm their strong relationship with the human assessment (with SIDE being the best in the class), while the conventional metrics struggle in capturing these quality aspects, with the only exception being the ROUGE-W-R when looking at the *conciseness*.

Still, the OR for SIDE is 1.3844 (*i.e.,* 38% higher odds of a unitary increment in the human assessment of summary conciseness for a one unit increase of SIDE), while for ROUGE-W-R is substantially lower (1.0954). The reason may lie behind the different assumptions made by the two families of metrics. To explain this point, let us take the example of the *conciseness* quality attribute. Metrics such as SIDE might learn during training that longer methods may need longer summaries and, thus, that very long summaries for short methods may not be appropriate. This is likely aligned with what a developer would think when judging the *conciseness* of a summary. Similarly, c_coeff is likely to produce low values when a very long summary is associated with a short method, since the denominator of the used formula (*i.e.,*

Table 10.2. RQ_2 - Ordered logistic regression model

Overall DA Score [RFA21]					
Metric	**OR**	**Value**	**Std. Error**	***t*-value**	***p*-value**
BLEU-1	0.9990	-0.0010	0.0017	-0.5822	0.6222
BERTScore-R	1.0029	0.0029	0.0023	1.2634	0.2943
SentenceBERT_CS	1.0057	0.0057	0.0020	2.8882	0.0100
InferNet_CS	0.9980	-0.0020	0.0025	-0.8064	0.5250
ROUGE-1-P	1.0058	0.0058	0.0018	3.2552	0.0033
ROUGE-4-R	1.0024	0.0024	0.0011	2.1987	0.04667
ROUGE-W-R	0.9996	-0.0004	0.0017	-0.2485	0.8040
c_coeff	1.0143	0.0142	0.0012	11.4549	<0.0001
CodeT5-plus_CS	1.0044	0.0044	0.0017	2.5383	0.0220
SIDE	1.0205	0.0203	0.0017	11.7029	<0.0001

Content Adequacy [RFA21]					
Metric	**OR**	**Value**	**Std. Error**	***t*-value**	***p*-value**
BLEU-1	0.9672	-0.0333	0.0347	-0.9615	0.4200
BERTScore-R	1.0525	0.0512	0.0468	1.0927	0.3929
SentenceBERT_CS	1.1121	0.1063	0.0406	2.6172	0.0225
InferNet_CS	1.0000	0.0000	0.0517	-0.0004	1.0000
ROUGE-1-P	1.0943	0.0901	0.0364	2.4733	0.0260
ROUGE-4-R	0.9914	-0.0086	0.0226	-0.3814	0.7811
ROUGE-W-R	1.0437	0.0428	0.0351	1.2202	0.3700
c_coeff	1.3729	0.3169	0.0255	12.4350	<0.0001
CodeT5-plus_CS	1.1413	0.1322	0.0358	3.6877	<0.0001
SIDE	1.6265	0.4864	0.0366	13.2942	<0.0001

Conciseness [RFA21]					
Metric	**OR**	**Value**	**Std. Error**	***t*-value**	***p*-value**
BLEU-1	1.0656	0.0636	0.0342	1.8576	0.1575
BERTScore-R	1.0160	0.0159	0.0470	0.3375	0.8770
SentenceBERT_CS	1.0587	0.0571	0.0403	1.4163	0.2617
InferNet_CS	1.0079	0.0079	0.0511	0.1547	0.8770
ROUGE-1-P	1.0075	0.0075	0.0359	0.2091	0.8770
ROUGE-4-R	0.9937	-0.0063	0.0228	-0.2766	0.8770
ROUGE-W-R	1.0954	0.0911	0.0346	2.6352	0.0267
c_coeff	1.2433	0.2178	0.0254	8.5798	<0.0001
CodeT5-plus_CS	1.0630	0.0611	0.0355	1.7212	0.1700
SIDE	1.3844	0.3253	0.0354	9.1835	<0.0001

Fluency [RFA21]					
Metric	**OR**	**Value**	**Std. Error**	***t*-value**	***p*-value**
BLEU-1	1.0650	0.0629	0.0345	1.8243	0.2267
BERTScore-R	1.0559	0.0544	0.0467	1.1638	0.4600
SentenceBERT_CS	0.9958	-0.0042	0.0404	-0.1043	0.9170
InferNet_CS	1.0543	0.0529	0.0515	1.0272	0.4600
ROUGE-1-P	1.0334	0.0329	0.0365	0.9004	0.4600
ROUGE-4-R	1.0304	0.0299	0.0229	1.3061	0.4600
ROUGE-W-R	0.9672	-0.0334	0.0347	-0.9603	0.4600
c_coeff	1.1680	0.1553	0.0253	6.1437	<0.0001
CodeT5-plus_CS	1.0249	0.0246	0.0357	0.6902	0.5444
SIDE	1.2826	0.2489	0.0359	6.9359	<0.0001

the number of terms in the summary) increases. Differently, conventional metrics would judge a generated summary of high quality if it is similar to the reference one. Thus, if the reference is not concise and the generated summary is similar to the reference (thus, not concise as well), the assessed quality will be high and not aligned with the developers' perception.

10.3.1 Qualitative Analysis

To provide further insights into the complementarity of the two families of metrics, we selected four qualitative examples to discuss (Fig. 10.2). Each example features (i) the Java method for which a summary was automatically generated; (ii) the target summary (*i.e.,* the one written by developers); (iii) the generated summary; and (iv) the quality scores assigned by the seven state-of-the-art (SOTA) metrics selected via the *redun* procedure (*i.e.,* `BLEU-1`, `BERTScore-R`, etc.), by our `SIDE` metric, and in the human assessment. We selected two examples in which `SIDE` provides an indication aligned with the developers' perception while the SOTA metrics fail, and two examples in which the opposite scenario occurs.

The first example ❶ is a typical scenario in which the reference summary is of low quality since it actually documents the fact that the method is "*not yet documented*".

The generated summary, instead, provides a good description of the method accordingly to the developers and, thus, is substantially different from the reference. This results in low values for the SOTA metrics, while `SIDE` agrees with the human assessment (*DA score* = 0.88), assigning the generated summary 0.91/1.00. In the second ❷ and third ❸ examples, `SIDE` is instead in disagreement with developers. In ❷, it assigns a high score (0.87) to a correct but very generic summary, while the conventional metrics are able to assess the low quality of the summary thanks to the presence of additional material present in the reference but lacking in the generated summary (*i.e.,* the example). In ❸, `SIDE` assigns a quite low score to the generated summary, despite it being well-judged by humans. SOTA metrics can instead exploit the word overlap between the generated and the reference summary, assessing the summary with higher scores. Finally, ❹ shows a low-quality generated summary (as assessed by developers) which is scored with the maximum scores by state-of-the-art metrics, since being identical to the reference one. `SIDE` is instead able to identify the summary as low quality (0.15) by contrasting it with the documented code.

In summary, our analysis suggests that a more comprehensive evaluation framework must be considered when assessing the quality of automatically generated summaries: Conventional metrics looking at the similarity between the generated and the reference summary are not enough and should be augmented by metrics judging the suitability of a generated summary for the code it documents, similarly to what `SIDE` does. The latter is the metric more related to the human judgment of code summary quality.

10.3.2 Ablation Study - Impact of Hard-negatives

In our replication package [repc], we present a table reporting the results of the ordered logistic regression model for each dependent variable including two variants of `SIDE`: One trained using hard-negatives (*i.e.,* the one related to the results discussed as of now) and one

METHOD 1

```
public Element asElement() {
    return this.component.createCopy();
}
```

REFERENCE SUMMARY

not yet documented

GENERATED SUMMARY

returns a copy of this component as an element

SCORES

SOTA							OUR	HUMAN			
BLEU-1	BertScore-R	SentenceBERT_CS	InferNet_CS	ROUGE-1-P	Rouge-4-R	Rouge-W-R	SIDE	DA	Adequacy	Concise	Fluency
0.34	*0.00*	*0.08*	*0.43*	*0.00*	*0.00*	*0.00*	*0.91*	*88*	*4*	*4*	*5*

METHOD 2

```
public String getId() {
    String val = "";
    String vals[] = doc.getValues("idvalue");
    for (int i = 0; i < vals.length; i++)
        val += vals[i] + (vals.length - 1 == i ? "" : " ");
    return val;
}
```

REFERENCE SUMMARY

gets the id for this record for example DLESE 000 000 000 001

GENERATED SUMMARY

returns the value of the element

SCORES

SOTA							OUR	HUMAN			
BLEU-1	BertScore-R	SentenceBERT_CS	InferNet_CS	ROUGE-1-P	Rouge-4-R	Rouge-W-R	SIDE	DA	Adequacy	Concise	Fluency
0.32	*-0.18*	*0.12*	*0.49*	*0.08*	*0.00*	*0.12*	*0.87*	*15*	*2*	*2*	*1*

METHOD 3

```
protected void handleReload(ArrayList additionalLocations) {
    fAdditionalLocations = additionalLocations;
    handleReload();
}
```

REFERENCE SUMMARY

handles reload of the list of additional locations

GENERATED SUMMARY

sets the given locations as additional locations to check then handles the reload

SCORES

SOTA							OUR	HUMAN			
BLEU-1	BertScore-R	SentenceBERT_CS	InferNet_CS	ROUGE-1-P	Rouge-4-R	Rouge-W-R	SIDE	DA	Adequacy	Concise	Fluency
0.52	*0.19*	*0.62*	*0.82*	*0.38*	*0.00*	*0.22*	*0.16*	*95*	*5*	*4*	*5*

METHOD 4

```
private void initialize() {
    textNumber = new Text(this, SWT.BORDER);
    super.putOnScreen(textNumber, 150, 19);
    labelNumber = new Label(this, SWT.NONE);
    labelNumber.setText("Number");
    ...
    this.setSize(new Point(354, 157));
}
```

REFERENCE SUMMARY

this method initializes this

GENERATED SUMMARY

this method initializes this

SCORES

SOTA							OUR	HUMAN			
BLEU-1	BertScore-R	SentenceBERT_CS	InferNet_CS	ROUGE-1-P	Rouge-4-R	Rouge-W-R	SIDE	DA	Adequacy	Concise	Fluency
1	*0.99*	*0.99*	*0.99*	*0.5*	*0.00*	*0.56*	*0.15*	*8*	*2*	*2*	*1*

Figure 10.2. Examples from Roy *et al.* [RFA21] dataset

trained without hard-negatives. In short, the impact of including or not hard-negatives in the training set is negligible, and the all findings of our study are not changed by such a design choice (*e.g.,* no impact on which ORs are statistically significant nor on their magnitude). This result may be due to several reasons. First, due to our definition of hard-negatives (*i.e.,* comments documenting at most 25% of the method's statements), we managed to create hard-negatives for only ∼15% of the methods in our dataset. Thus, it is possible that a higher coverage is needed to observe any influence on the achieved results. Second, our definition of hard-negatives may be suboptimal, opening to better approaches aimed at creating hard-negative samples for contrastive learning applied to software engineering tasks.

10.4 Threats to Validity

Construct validity threats. As also detailed in the work of Roy *et al.* [RFA21], the considered dependent variables reflect the user's assessment of a summary from different perspectives (*conciseness*, *fluency*, *content adequacy*, plus an *overall assessment*). We are aware that such an assessment could suffer from the assessor's error or subjectiveness. Also, we acknowledge that the fine-tuning dataset used to train SIDE may feature some low-quality summaries which may partially hinder its ability to assess whether a natural language text represents a suitable summary for a given code. However, our empirical evaluation showed that SIDE is still the metric better capturing human assessment of code summary quality, despite this potential source of noise in its training set.

Internal validity threats. These primarily pertain to the configurations applied during metric computation and evaluation, such as the decision to utilize cosine similarity. In addition, we recognize the potential value in exploring alternative hyperparameter settings (see Section 10.1.4) when developing SIDE.

Also, adopting CodeT5+ as the representative state-of-the-art pre-trained model for generating $\langle method, documentation \rangle$ embeddings, and subsequently using cosine similarity for comparison, might not reflect the best strategy for the discussed task.

Finally, the negligible impact of including hard-negative samples in the training set may be due to the design we employed to mine hard-negatives (*e.g.,* see the choice of the 25% coverage to identify hard-negative comments) and further studies are needed to investigate alternative solutions.

Conclusion validity threats. RQ_1 findings show both the metrics correlation, computed using a non-parametric procedure, and the PCA. RQ_2 is based on an ordered logistic regression, suitable for variables in ordinal scale. Note that statistically significant p—values are small enough to be unaffected by fishing and error rate.

External validity threats. The main limitation of our study is that it relies on human evaluations from a single dataset [RFA21]. Nevertheless, such a dataset is large and features a total of 6,253 (5,201 used in our study) evaluations from 226 different participants.

Concerning the independent variables (evaluation metrics), our study attempted to consider several metrics having a different nature, some of which were used in previous work [HEBM22, RFA21, SHJ13].

10.5 Conclusions

State-of-the-art metrics used to evaluate code summarization techniques only assess the similarity between the generated summary and the reference one written by developers. This implies that a generated summary, while suitable for the code to document, may be different from the reference one (*e.g.,* a low-quality one), and therefore scored low by these metrics. We argue that the code to document must also be considered in the equation when scoring automatically generated summaries, to assess whether they are suitable for such a code independently from their similarity with the reference summary. To capture this information, we presented SIDE, a metric exploiting contrastive learning to learn characteristics of suitable and unsuitable summaries for a given code.

We run a study involving 40 metrics (including SIDE) investigating their complementarity and the extent to which they correlate with humans' evaluation of summary quality. Our findings highlight the high complementarity between SIDE and the metrics focusing on the similarity between the generated and the reference summary. SIDE is also the metric that better describes humans' assessment of summary quality. Also, we show that the proposed contrastive learning-based metric captures the suitability of a summary for a given code better than simpler solutions. Further evaluations of code summarization methods should incorporate a broader assessment framework that includes both conventional metrics—*e.g.,* the similarity between generated and reference summaries—and SIDE-like metrics, evaluating the suitability of the generated summary for the code being documented. In addition, the findings of this study provide several opportunities for researchers, particularly in the area of program comprehension. To this end, a key application of SIDE would involve the identification of inconsistencies between code comments with respect to the documented code (*e.g.,* a method), as this could significantly improve the quality of software systems once the code and its documentation are realigned.

Part V

Epilogue

11

Conclusions and Future Work

In this thesis, we investigated the viability of utilizing pre-trained deep learning (DL) models for code-related tasks, with a specific focus on tasks requiring the manipulation of bi-modal data, such as code summarization and the generation of log statements. Our preliminary experiments with pre-trained models, documented in Chapter 3, were among the first in the literature showing the potential of pre-trained models for code related tasks and contributed to pave the way to several subsequent works presented by other research groups [TDSS22, ZJW+23, NSM23, VPSO23, FCG+24, WWJH21].

We also took advantage of the achieved positive results to propose novel techniques automating tasks such as code snippet summarization (Chapter 8), code comment completion (Chapter 9), and log injection (Chapter 6), by also learning about the difficulties in automatically assessing the correctness of the recommendations generated by these models. This led us to the proposal of a novel metric to evaluate the quality of automatically generated code summaries (Chapter 10).

Finally, we studied the robustness of the state-of-the-art code recommender GitHub Copilot in the task of code generation (Chapter 4), also characterized by bi-modal data, with code that must be generated starting from a natural language description.

In the following we discuss the limitations of our work and future research directions.

11.1 Limitations and Future Work

11.1.1 Applicability and generalizability of our findings across various programming languages and models

Our research was carried out using *Java* as the primary programming language, meaning that the DL-based methods we devised were initially pre-trained and subsequently fine-tuned for this programming language. While we expect our findings to generalize to other languages, additional empirical investigations are needed to verify such a broad applicability.

Furthermore, our studies were framed within the context of sequence-to-sequence Transformer models, specifically the T5 architecture, which comprises two distinct parts: (i) an

encoder and (ii) a decoder, each based on separate Transformer networks and linked together. It would be interesting to explore whether our insights remain applicable across different model architectures, such as those that rely solely on a decoder.

11.1.2 Evaluating the perceived usefulness of our techniques

The evaluations of our techniques are mostly based on predictions performed by the trained models on a given test set. This means that the actual usefulness of the techniques from the developers' perspective has not been experimented. This is part of our future work, especially for techniques for which we also released an usable tool implementing them. An example of this is JLOG [jlo], a tool made available for Visual Studio Code, developed on the basis of LEONID (Chapter 6). Our goal is to exploit this tool to perform an "in-vivo" study with developers to assess the actual benefits (if any) brought by such a tool as perceived by developers.

11.1.3 In-context learning and prompt engineering for software-related practices

Large Language Models (LLMs) have recently showcased capabilities beyond their original applications in natural language comprehension and generation. Key to this adaptability are two groundbreaking approaches: "in-context learning" [DLD+22] and "prompt engineering" [WFH+23] that have empowered these models to adapt and excel in a diverse array of domains and applications.

Within this framework, we plan to investigate how in-context learning and prompt engineering can be further tailored to understand and generate code, automate documentation, and (more in general) software engineering practices. This entails exploring the integration of LLMs into the development environment to provide real-time assistance and feedback to developers. We intend to undertake this analysis specifically for those areas of software engineering where our research has indicated that LLMs do not meet expectations. Notably, our recent work into CI/CD (Continuous Integration/Continuous Development) pipelines (see *Appendix:*Chapter B), and the management of SATD (Self-Admitted Technical Debt) (see *Appendix:*Chapter A) provide critical insights. In both scenarios, we observed that the "pretrain-then-finetune" approach remains superior, surpassing the performance of advanced AI-driven tools such as Copilot [cop] and ChatGPT [cha] when assisting practitioners for the completion of Workflow files and the automated repayment of SATD.

11.1.4 Green-AI for software engineering

In Chapter 3 –Section 3.3.4 we presented the overall training time required to perform pretraining and fine-tuning of the T5-based solution we developed. It took 175h, which equates to 126.85kg CO2e emitted in the atmosphere. To put this number into perspective, it would be the same amount of CO2e that a single person would emit flying 2.5 times from Paris to London.[1]

[1]The computation is performed using a publicly available calculator for energy consumption: `http://calculator.green-algorithms.org`

Referencing the specific details of the T5 model employed in our research, we selected its smallest configuration, which features ∼60M parameters. In contrast, examining LLMs such as GPT-3, which powers [cha], the training of this model demanded 14.8 days on 10,000 V100 GPUs [PGL+21]. The CO2e emissions attributed to this process amounted to 502 metric tons, comparable to the emissions from driving 112 gasoline-powered vehicles a year.[2].

Despite the substantial environmental impact associated with training these LLMs, the inference process (*i.e.,* prompting) has been projected to become the primary contributors to carbon emissions, as indicated by recent research [dV23, WRG+22, CLN+23]. In light of this and the increasing reliance on LLMs for automating software engineering practices, our future work will explore strategies for incorporating specific software engineering principles into LLMs prompting, with the aim to reduce the environmental footprint of utilizing such models for software engineering tasks.

11.2 Closing Words

Our research aimed to enhance the automation of various code-related tasks, addressing shortcomings of existing techniques, also considering the limitations in the empirical evaluations reported in the literature (*e.g.,* inappropriate metrics for code summarization). Given the fast pace at which software engineering automation is changing thanks to the advent of general-purpose LLMs, it remains challenging to predict how long-lasting many of our findings will be. Also, as we showed, some of the novel AI-based solutions that are being introduced in the developers' workflow (*e.g.,* Copilot) pose novel challenges for developers, such as those of crafting a proper prompt to maximize the usefulness of these tools. This leaves us with the question of how software engineering will change in the near future, and how developers will need to adapt in this context. These are questions that will inform our future research.

[2] https://news.climate.columbia.edu/2023/06/09/ais-growing-carbon-footprint/

Appendices

In the appendix, we provide further details on contributions to Software Engineering that are outside the main focus of this thesis.

Appendix A presents our investigation into managing Self-Admitted Technical Debt (SATD) and explores the potential for its automated repayment through advanced DL-based approaches.

> **Towards Automatically Addressing Self-Admitted Technical Debt: How Far Are We?**
> Antonio Mastropaolo, Massimiliano Di Penta, Gabriele Bavota. In *Proceedings of the 38th ACM International Conference on Automated Software Engineering (ASE 2023)*, pp. 585-597

Appendix B describes GH-WCOM (GitHub Workflow COMpletion) an approach able to assist developer in completing GitHub Workflow files:

> **Toward Automatically Completing GitHub Workflows**
> Antonio Mastropaolo, Fiorella Zampetti, Massimiliano Di Penta, Gabriele Bavota. In *Proceedings of the 46th ACM/IEEE International Conference on Software Engineering (ICSE 2024))*, pp. 1-12

Appendix C outlines an empirical investigation into the performance of three data-driven techniques in supporting automated variable renaming:

> **Automated variable renaming: are we there yet?**
> Antonio Mastropaolo, Emad Aghajani, Luca Pascarella, Gabriele Bavota. In *Proceedings of Journal Empirical Software Engineering (EMSE 2023)*, Volume 28(2)

Appendix D details an empirical study aimed at examining how developers use ChatGPT to automate and support software engineering practices:

> **Unveiling ChatGPT's Usage in Open Source Projects: A Mining-based Study**
> Rosalia Tufano and Antonio Mastropaolo, Federica Pepe, Ozren Dabic, Massimiliano Di Penta, Gabriele Bavota. In *Proceedings of the 21st International Conference on Mining Software Repositories (MSR 2024)*, To appear, 12 pages

A

Towards Automatically Addressing Self-Admitted Technical Debt: How Far Are We?

Technical Debt (TD) has been defined by Cunningham as "not-quite-right code" [Cun92]. Essentially, the terms refer to the debt an organization or an individual developing and releasing software should repay to make it acceptable, for example in terms of functionality, reliability, or maintainability. Oftentimes, developers achieve awareness of the TD in their program, admitting it through comments, commit messages, or issues. This has been referred to as "Self-Admitted Technical Debt" (SATD) [PS14b]. Previous research has investigated why developers annotate software as SATD, essentially to keep track of what needs to be improved [ZFSD21]. Also, the analysis of SATD in existing programs has shown how developers take it seriously, as it gets removed in the majority of the cases [dSMASS17], even though this often happens when the source code is completely replaced or even removed [ZSD18]. As previous work found [BR16, FCZ+21, PS14b, ZFSD21], SATD relates to different problems in the program, such as the need to fix bugs occurring in certain circumstances, enhancing or even completing a feature, or improving the source code maintainability and quality in general.

Researchers have proposed various kinds of approaches to aid developers with the management of SATD. On the one hand, while SATD comments are usually recognizable by commonly used keywords such as "TODO" or "FIXME", this is not always the case. Therefore, approaches leveraging several types of techniques ranging from simple regular expression matching [PS14b] to shallow machine learning [dSMST17], and deep learning [RXX+19b] have been used to identify SATD comments.

Also, some approaches recommend developers with the type of change that needs to be carried out to address the SATD [ZSD20]. While previous work has proposed approaches to guide developers towards paying back TD, to the best of our knowledge there is no specific work aimed at automatically resolving (SA)TD. Indeed, this can be a direction worthwhile to investigate, considering the significant advances in the application of generative approaches to software-related tasks, such as code completion [CCP+21, SLH+21, SDFS20, LWLK17], program repair [MH21a, JLT21, LPP+20b, LWN20], vulnerability patching [CKM22, FTL+22a], or code review [TPT+21, TMM+22, LLG+22b, LLG+22a]. However, given the diversity of

SATD, its repayment would require approaches that go beyond what has been already devised for each of the aforementioned tasks. Therefore, this paper aims to answer the following question:

Can AI-based approaches automatically repay the technical debt?

To answer this question, we investigate the extent to which neural generative approaches based on deep learning transformers [VSP+17] can be used to repay SATD. An obvious question that can arise is whether SATD resolution is, in the end, equivalent to program repair. We believe there are a series of differences and challenges:

1. As also pointed out by previous work [ZFSD21], repaying (SA)TD not only means fixing bugs, but it also requires to improve the code in different ways, for example enhancing a feature, making the code able to handle certain special scenarios, or adopting alternative APIs.
2. Differently from other "buggy" code, SATD admits the presence of a problem, therefore directly highlighting the portions of the source code that need to be repaired.
3. Neural models require a conspicuous amount of training data, which may be available for certain tasks (*e.g.,* code completion), and less for others, including SATD resolution.

To study how SATD can be addressed by generative models, we first created a dataset of 5,039 SATD removal instances from 595 open-source projects by leveraging an available tool that detects SATD removals [ACB+22]. Then, we leverage a pre-trained transformer model (CodeT5 [WWJH21]) that previous work showed particularly effective to cope with problems where the size of the training set is limited [FTL+22a, WYG+22]. Through different experiments, we test the effectiveness of several pre-training/fine-tuning strategies, including:

1. No pre-training, fine-tuning using SATD removal instances.
2. Self-supervised pre-training followed by fine-tuning on SATD removals. Self-supervised pre-training exploits training objectives not requiring a supervised dataset. We use the *masked language model* objective [DCLT19, LOG+19], which consists in providing the model with input sentences (*e.g.,* an English sentence, a *Java* method, depending on the language of interest) having 15% of their tokens masked, asking the model to predict them.
3. Self-supervised and supervised pre-training followed by fine-tuning on SATD removal instances. Supervised pre-training can be used to pre-train the model on a task similar to the downstream one. In our case, we pre-train the model for the implementation of generic code changes, before fine-tuning it with SATD removals.

We also experimented with the impact on the model's performance with and without the SATD comment. This is worthwhile to study because (i) the SATD comment may act as a sort of prompt-tuning for the transformer [VSP+17] which has been shown to help models pre-trained on English text (such as CodeT5); and (ii) a boost in performance motivates the usefulness of SATD comments not only as a trace for developers [ZFSD21], but also as a way to aid AI-based approaches. Finally, we experiment with the extent to which SATD can be addressed by leveraging a Large Language Model (LLM) chat bot, *i.e.,* ChatGPT [cha].

Results of our study indicate that automatically addressing SATD instances is a challenging task, and the best model we experimented with is able to correctly address 2.30% (one attempt) to 8.10% (ten attempts) of the SATD instances in our test set. Without any pre-training, the model is not able to address any SATD instance, likely due to the limited size of the fine-tuning dataset. Self-supervised pre-training helps in improving performance, which is further increased when the model is also subject to supervised pre-trained on a task (*i.e.,* implementing generic code changes) resembling the downstream one (*i.e.,* addressing SATD). Finally, we experimented with three different prompts for ChatGPT, with the best one being able to address only 1.19% of the SATD in our dataset, confirming how challenging the tackled task is. In summary, while the studied approaches can in some cases automatically repay SATD, there is still a long way to go to fully address this problem.

Overall, the paper contributes to the state-of-the-art on (SA)TD management and resolution with:

1. An experimentation featuring seven different combinations of treatments on the use of pre-trained neural transformers for SATD repayment;
2. Results of a study on the feasibility of using a LLM chat bot (ChatGPT) for SATD repayment; and
3. A replication dataset that can be also used for further experiments in this area [repf].

A.1 Study Definition, Design and Planning

The *goal* of this study is to evaluate DL-based solutions in automatically implementing code changes required to address SATD in *Java* code. The *context* of the study features two deep learning models, namely CodeT5 [WWJH21] and ChatGPT [cha], and two datasets used for pre-training and fine-tuning the experimented models. The pre-training dataset features generic code changes implemented by developers in open-source projects and has been presented in the work by Tufano *et al.* [TPW+19]. The fine-tuning dataset is a contribution of this paper and features SATD removal changes.

In the following, we formulate the study research questions (RQs) (Section A.1.1). Then, Section A.1.2 describes the datasets used to train and test the experimented techniques. The latter are presented in Section A.2. We conclude by outlining the data analysis procedure in Section A.3.

A.1.1 Research Questions

Given our overall goal (*i.e.,* assessing the capabilities of DL-based solutions in automatically addressing SATD), we formulate the following RQs:

RQ_1: *To what extent do pre-trained models of code support automated SATD repayment?* In RQ1 we fine-tune the pre-trained CodeT5 [WWJH21] model for the task of SATD repayment and assess its performance. We also investigate the role played by the self-supervised pre-training on CodeT5, *i.e.,* the extent to which the self-supervised pre-training helps in fixing SATD. RQ1 provides a starting point for our investigation, showing what performance can be achieved by just fine-tuning an existing pre-trained model for SATD repayment.

RQ_2: *To what extent does the infusion of "similar-task knowledge" in pre-trained models of code benefits the automated SATD repayment?* While CodeT5 [WWJH21] has been pre-trained using the *masked language model* self-supervised objective, other forms of pre-training are possible. RQ_2 evaluates the effectiveness of performing a further pre-training step aimed at instilling in the model knowledge about a task resembling the downstream one (*i.e.,* the SATD repayment). This means that, before fine-tuning the model for SATD repayment, we leverage a supervised pre-training in which the model learns how to implement generic code changes. The rationale is that this task, while different from SATD repayment, can start driving the model's weights toward a configuration closer to the one needed for the downstream task.

RQ_3: *To what extent does the presence of "context-specific knowledge" help pre-trained models of code in the automated SATD repayment?*

SATD instances can be represented as pairs $\langle comment, code \rangle$ where the $comment$ describes the SATD to address in the $code$. When assessing the performance of DL-based solutions in automatically addressing SATD, the $\langle comment, code \rangle$ pair represents the input of the model which is expected to produce a $revised\_code$ addressing the technical debt. If we factor out the $comment$ from the input the task becomes similar to those previously tackled in the literature through DL models, such as learning generic code changes implemented by developers [TPW+19] or fixing bugs [TWB+19a, JLT21, MH21a, LPP+20b, CKM22]. For these tasks the model's input is just a $code$ in which a change (*e.g.,* a bug fix) must be implemented, thus producing as output the $revised\_code$. RQ_3 assesses the extent to which providing the SATD $comment$ to the model helps to address the TD, thus investigating whether training a SATD-specialized model is worthwhile as compared to just using a model trained to address generic code changes without relying on the SATD comment (see *e.g.,* [TPW+19]).

RQ_4: *Are general-purpose large language models zero-shot learners for SATD repayment?* In RQ4 we study whether LLMs (and, in the specific case, ChatGPT [cha]) can be considered as out-of-the-box solutions for the automated SATD repayment. While, for what concerns source code ChatGPT has been "seen" the entire GitHub, it has not been fine-tuned for the specific problem of SATD repayment. A positive answer to RQ4 would indicate that research on specialized models for SATD repayment is unlikely to be relevant/beneficial.

A.1.2 Context: Datasets

We describe the pre-training and fine-tuning datasets used in our research, which are summarized in Table A.1.

A.1.2.1 Self-supervised pre-training on bi-modal data

In the first three RQs, we employ CodeT5 [WWJH21] as representative of a state-of-the-art pre-trained model of code. CodeT5 is a Text-To-Text Transfer Transformer (T5) model [RSR+20] pre-trained on code and natural language (*i.e.,* code comments). Among different transformer models we have chosen CodeT5, as it has been used successfully for several tasks, including code summarization [WWJH21], source code generation [ZAX+22], vulner-

ability patching [FTL+22a], code review automation [LLG+22a], code-to-code translation [KABV22], and shown to outperform other models such as CodeBERT [FGT+20], PLBART [ACRC21] and GraphCodeBERT [GRL+21].

Wang *et al.* [WWJH21] utilized the CodeSearchNet dataset [HWG+19] for pre-training CodeT5 using the masked language model objective (*i.e.,* self-supervised pre-training by randomly masking 15% of the input asking the model to predict it). This dataset includes functions written in six different programming languages (Go, Java, JavaScript, PHP, Python, and Ruby). A subset of these functions also includes a top-level comment (*e.g.,* Javadoc for Java). In addition, Wang *et al.* gathered extra data from C/C# repositories hosted on GitHub. This led to a total of 8,347,634 pre-training code functions: 3,158,313 of these functions are paired with their documentation, while 5,189,321 consist solely of code.

A.1.2.2 Supervised pre-training on generic code changes

Tufano *et al.* [TPW+19] proposed the usage of NMT to learn how to automatically apply code changes implemented by developers during pull requests (PRs). The NMT model has been trained on a dataset featuring PRs from three *Gerrit* [ger] repositories: (i) *oVirt*, (ii) *Android*, and (iii) *Google*.

Each of these repositories groups multiple projects (*e.g.,* all those related to the *Android* operating system), with the authors focusing on the ones written in *Java*. For each PR, Tufano *et al.* extracted two versions of the files involved in the change diff: The version before the PR was implemented and the version after the PR has been merged. These files have then been parsed to extract a total of 631,307 pairs of methods $\langle m_b, m_a \rangle$ representing the same method before (m_b) and after (m_a) the changes implemented in the PR. The idea was to train the NMT model on these pairs to see if it was able to learn generic code changes that developers might implement in the context of PRs.

We leverage this dataset in the context of RQ_2 to investigate whether the infusion of "similar-task knowledge" in pre-trained models of code benefits the automated SATD repayment. Starting from the ∼630k pairs in the original dataset we discarded instances containing non-ASCII tokens, and those having $\#tokens > 1024$. The latter filter is necessary to manage the computational complexity of training large DL-models and is a common practice in the software engineering literature [MPB22, CCP+21, TMM+22, TDS+20, TPT+21, TPW+19]. For example, in the original work by Tufano *et al.* [TPW+19] in which this dataset has been created, the authors removed all pairs having $\#tokens > 100$. Subsequently, we eliminate duplicated pairs $\langle m_b, m_a \rangle$, obtaining a final dataset of 284,190 instances. We split the processed set of methods into 90% training and 10% validation. The former will be used to perform supervised pre-training on the task of learning generic code changes. The latter is instead employed to identify the best-performing checkpoint while performing the supervised pre-training.

A.1.2.3 Fine-tuning dataset of SATD removals

As a first step to build the fine-tuning dataset, we collected a list of *Java* repositories leveraging the GitHub Search tool by Dabic *et al.* [DAB21]. The querying user interface allows

Table A.1. Number of instances included in the datasets we used to train, test and evaluate the models

Dataset	Train	Test	Eval	Overall
Generic code changes [TPW+19]	255,771	-	28,419	284,190
SATD removal	3,537	1,000	502	5,039
Total	259,308	1,000	28,921	289,229

to identify GitHub projects that meet specific selection criteria. We selected all *Java* projects having at least 500 commits, 10 contributors, 10 stars, and not being forks (to reduce the chance of mining duplicated code). The commits/contributors/stars filters aim at discarding personal/toy projects. Instead, the decision of narrowing down the scope to only *Java* as a programming language was dictated by (i) the will of reusing the previously described *Java* dataset by Tufano *et al.* (Section A.1.2.2) for supervised pre-training, and (ii) the usage in our toolchain of tools only supporting Java (*e.g., SATDBailiff*, as described in the following). We collected 6,971 *Java* repositories.

To extract changes aimed at addressing SATD instances we rely on the *SATDBailiff* tool by AlOmar *et al.* [ACB+22]. *SATDBailiff* can identify commits featuring the addition, removal, or change of SATD instances in the history of *Java* git repositories. We are interested in mining from the history of the subject *Java* repositories commits featuring removal events, since those are the ones implementing changes aimed at addressing SATD and, thus, are the ones suitable for our fine-tuning. Unfortunately, we found this extraction process to be extremely expensive. For this reason, we set two boundaries for our data mining. First, we allowed *SATDBailiff* to process each repository for at most 60 minutes. Indeed, we observed that for some very large projects the mining procedure could go on for days. If a repository could not be analyzed within 60 minutes, the repository was discarded from our study. Second, we set 50 days as the maximum boundary for the mining procedure. In this period, *SATDBailiff* successfully analyzed the entire history of 809 *Java* projects hosted on GitHub.

These 809 *Java* projects yielded a total of 519,440 SATD-related events including *SATD_REMOVED*, *SATD_ADDED*, and *SATD_CHANGED*. We extracted the 139,803 concerning *SATD_REMOVED*. Then, we further refined this list by only selecting instances for which the SATD comment extracted by *SATDBailiff* contained specific keywords describing the presence of *Self-Admitted Technical Debts*: `to(-)do`, `fix(-)me`, `check(-)me`, `hack(-)me`, and `xxx`. In principle, we are aware that such further filtering may reduce the dataset construction recall and, ultimately, the dataset size. However, as the SATD detection performed by *SATDBailiff* is based on a ML-based approach, it is inherently subject to false positives, and we wanted to avoid experimenting with changes that were not related to SATD removal. We have chosen the aforementioned keywords since those are well-known patterns used to signal SATD [PS14b, CZN+22]. After this further pruning, we obtain a dataset where each instance is a triplet `commit_before`, `commit_after`, and `comment`, where the two commits indicate the version of the code affected (`commit_before`) by and cleaned from (`commit_after`) the SATD, while `comment` is the code comment documenting the SATD which is linked to a specific *Java* file.

Msatd types featured within our SATD Removal Dataset

```java
@Override
//TODO: need to support compound indexes
public void ensureIndex(final String key, final OrderBy orderBy)
{
    ensureIndex(key, orderBy, false);
}
```

1

```java
private void configureOptions() {
   LOG.debug("Auto detecting current execution environment");
   // FIXME Make it pluggable
   this.options.putExtraAttribute(
                FileIoProcessor.OPTION_EXPORTER_ENABLED,
                GenericOptionValue.AUTO.getSymbol());
}
```

2

Figure A.1. SATD removal instances in our fine-tuning dataset

We removed duplicated instances, namely those being characterized by the same triplet (*e.g.,* due to the same SATD fixed in the same commit in different files). This left us with 75,083 instances which could feature the SATD in any part of the impacted *Java* file. However, we are interested in training the DL-model to address SATDs affecting a specific method, ignoring SATD instances related to *e.g.,* class instance variables, `import` statements etc. The reason for such a choice is two-fold. First, also the pre-training datasets are defined at function-level granularity, thus suggesting a similar fine-tuning to take full advantage of the knowledge acquired during pre-training. Second, providing an entire *Java* file as input to the model makes the training extremely expensive, since the length of the input sequences will grow to tens of thousands of tokens. Thus, we parsed the code file in the `commit_before` to see if the SATD comment was within a method m_{satd} or immediately above it. This was the case for 65,380 instances.

Scenario ❶ depicted in Fig. A.1 shows a SATD comment preceding the implementation of `ensureIndex` method, while in ❷ the comment documenting the TD is included within the body of the `configureOptions` method. Based on what we have explained so far, we work on the following assumption:

> If an SATD comment is removed, the changes performed within the same commit and in the method to which the SATD comment was attached are related to repaying such an SATD.

Given the available dataset, our aim is to create the final training triplets $\langle m_{satd}, m_{fixed}, comment \rangle$, where $\langle m_{satd}, comment \rangle$ represents the model's input and m_{fixed} the model's output. To achieve this goal, some additional checks are required. First, as previous work has found [ZSD18], it is possible that addressing the SATD requires the deletion of m_{satd}, or the implementation of other methods while leaving m_{satd} unchanged (except for the removal of the SATD comment). Thus, we verify that (i) a method having the exact same name of m_{satd} exists in `commit_after`, and (ii) by removing the SATD comment from m_{satd} we obtain a method $m'_{satd} \neq m_{fixed}$. The first filter guarantees that m_{satd} still exists in `commit_after`, while the second ensures that changes have been implemented in m_{satd} to address the SATD (obtaining m_{fixed}). This cleaning left us with 12,267 triplets.

We manually inspected 100 triplets to look for additional problematic cases. We found triplets characterized by "meaningless" SATD comments, such as "TODO" not followed by anything else. These comments do not really describe a SATD and, for this reason, we decided to exclude from our dataset all triplets having a $comment$ featuring less than three words that are unlikely to describe a SATD in enough detail to be understood (8,564 instances left).

Finally, we used the code-tokenize Python library [CR22] to extract a tokenized version of the extracted methods and removed triplets featuring methods having $\#tokens > 1024$, and instances that raised errors while being tokenized. After filtering, we ended up with a total of 5,039 triplets derived from 595 *Java* projects, which constitute our fine-tuning dataset. The latter is further split into 70%, 20%, and 10% for training, testing, and validation of the models, respectively. These triplets have been processed to introduce two special tokens $\langle SATD\_START \rangle$ and $\langle SATD\_END \rangle$ which serve to tag the start and end of the SATD comment within m_{satd}. As previous work did [FTL+22a, TMM+22], the idea is that these tokens could help direct the model's attention toward relevant sections of the input. Fig. A.2 depicts an example of instance from the dataset we built.

Note that we create two different versions of the SATD removal dataset, both containing the same number of instances across training, testing, and validation.

However, while the first one also contains the SATD comment, the latter is removed in the second one. This is necessary to address RQ_3, *i.e.,* to determine the extent to which admitting TD would not only serve as a trace for the developers but also as an aid for automated tools.

A.2 Experimented Techniques

As we are interested to study the performance of different DL-based solutions for automatically addressing SATD instances, we focus on two recently presented models, namely CodeT5 [WWJH21] and ChatGPT [cha]. For the former, we use the $CodeT5_{base}$ variant, featuring 220 million trainable parameters. We use the default architecture and hyperparameters of $CodeT5_{base}$ featuring 12 Transformer Encoder blocks, 12 Transformer Decoder blocks, 768 hidden sizes, and 12 attention heads. The learning rate is set to 2e-5.

While CodeT5 has been specifically pre-trained and fine-tuned to support software engineering tasks, ChatGPT is a general-purpose LLM designed and developed by OpenAI to produce human-like responses for a broad spectrum of language-related tasks (*e.g.,* question-answering, language translation, coding tasks, etc.). There are currently two versions of ChatGPT, one built on top of GPT-3.5 and one exploiting GPT-4.0. Given the current restrictions on the usage of GPT-4.0, we carried out our research using GPT-3.5 as the foundational model for ChatGPT. Although considered "less proficient" than the chat-bot built using GPT-4.0, the version employed for our experiments, with 154 billion parameters model (GPT-3.5), is still in the LLM category, and it is definitely by far a larger model than CodeT5.

In the following, we detail how we used the two models to answer our RQs. In particular, we pre-trained, fine-tuned, and queried the models using several different strategies. All fine-tunings have been executed for a maximum of 50 epochs on the "SATD removal" dataset (see Table A.1). To cope with overfitting, we stop the fine-tuning using an early stopping

METHOD INTRODUCING THE SATD COMMENT (M_{satd})

```
public void beforeTest(TestContext context) {
    if (beforeTest != null) {
       for (SequenceBeforeTest sequenceBeforeTest : beforeTest) {
          try {
             if (sequenceBeforeTest.shouldExecute(getName(),
                    getPackageName(), null))
   <SATD_START> //TODO provide test group information <SATD_END>
                    sequenceBeforeTest.execute(context);
           }
           catch (Exception e) {
             throw new CitrusRuntimeException("Before test…", e);
           }
       }
    }
}
```

METHOD RESOLVING THE SATD COMMENT (M_{fixed})

```
public void beforeTest(TestContext context) {
    if (beforeTest != null) {
       for (SequenceBeforeTest sequenceBeforeTest : beforeTest) {
          try {
             if (sequenceBeforeTest.shouldExecute(getName(),
                    getPackageName(), groups ))
                    sequenceBeforeTest.execute(context);
           }
           catch (Exception e) {
             throw new CitrusRuntimeException("Before test…", e);
           }
       }
    }
}
```

Figure A.2. SATD removal instance in our fine-tuning dataset

procedure assessing the loss of the model on the validation set every epoch, using a delta of 0.01 and patience of 5. This means that the training process stops if a gain lower than delta (0.01) is observed after 5 consecutive epochs and the best-performing checkpoint up to that training step is selected.

A.2.1 No Pre-training + Fine-tuning (RQ_1)

We fine-tune on the context-specific SATD removal dataset (*i.e.,* the one including the comment documenting the SATD) a $T5_{base}$ model [RSR+20] (*i.e.,* the same used for CodeT5 [WWJH21]) without any pre-training. To this aim, we start by randomly initializing the weights of the model, which will be adjusted during the fine-tuning procedure. Such a model serves as a baseline to assess the impact of different pre-trainings on the model's performance.

A.2.2 Self-supervised Pre-training + Fine-tuning (RQ_1)

We start from the CodeT5 model pre-trained using a self-supervised objective (*i.e., masked language model*) and fine-tune it on the context-specific SATD removal dataset.

A.2.3 Self-supervised & Supervised Pre-training + Fine-tuning (RQ_2 and RQ_3)

Previous works in the natural language processing [ZZSL20a] and in the software engineering literature [CKM22, WLX+19] suggest that exploiting a supervised pre-training objective that resembles the downstream task (in our case, SATD repayment) can play a positive role on the models' performance. For this reason, we further pre-trained CodeT5 for five epochs using the "generic code changes" dataset described in Section A.1.2.2. Following that, we continue fine-tuning CodeT5, which has been enhanced with domain-specific knowledge, on the SATD removal dataset, both with and without additional context, *i.e.,* code comments.

A.2.4 Zero-Shot Prompt Tuning (RQ_4)

We designed three prompt templates aimed at querying ChatGPT for the SATD repaying task. All prompts feature the SATD $comment$ and the method affected by SATD (m_{satd}):

1. `Remove this SATD: {`$comment$`} from the following code {`m_{satd}`}`
2. `Perform removal of this SATD: {`$comment$`} from this code {`m_{satd}`}`
3. `This code {`m_{satd}`} contains the following SATD: {`$comment$`} remove it`

We also tried to explain ChatGPT the notion of SATD before querying it with any of the three above-listed prompts. However, we did not observe significant changes in the output, thus indicating that ChatGPT is "aware" of what SATD is.

A.3 Data Collection and Analysis

We run each trained CodeT5 on the 1,000 *Java* methods in the test set, asking it to implement the code changes needed to repay the SATD. We use the beam search decoding schema [FAO17] to produce multiple candidate repayments for an input m_{satd}. In the case of ChatGPT, we use the OpenAI APIs to query it. However, the APIs do only allow collecting a single answer (solution) from ChatGPT.

In the following, we summarize the seven different models' configurations we experiment with:

- 1: *No pre-training + context-specific fine-tuning*, in the results referred as M0;
- 1: *CodeT5 + context-specific fine-tuning*, referred as M1;
- 1: *CodeT5 + supervised pre-training on code changes + context-specific fine-tuning*, referred as $M2_{CC}$;
- 1: *CodeT5 + supervised pre-training on code changes + no-context fine-tuning*, referred as $M3_{CC-Ablation}$;
- 3: *ChatGPT in zero-shot learning setting* × 3 prompt templates ($M4_{T1-T3}$), where the digit 1-3 indicates the used template among those described in Section A.2.4.

We assess the performance of each model using two metrics. First, the percentage of Exact Match predictions for different beam sizes K (EM@K), namely the cases in which the generated output is identical to the expected m_{fixed}. For CodeT5 we experiment with

K equal 1, 3, 5, and 10. For the reasons previously explained, we only computed EM@1 for ChatGPT. Second, we compute the CrystalBLEU score [EP22] between the generated predictions and the m_{fixed} target. CrystalBLEU measures the similarity between a candidate (predicted code) and a reference code (oracle), similar to how the BLEU score [PRWZ02] measures similarity between texts. However, CrystalBLEU is specifically designed for code evaluation, while retaining desirable properties of BLEU, specifically being language-agnostic and minimizing the effect of trivially shared n-grams, which would produce inflated results.

To better understand the extent to which the considered techniques can successfully address SATD, we analyze the edit actions (*i.e.,* deleting, adding, moving, or changing) to code elements required in each SATD repayment instance. To this aim, we use the Gumtree Spoon AST Diff [FMB+14] to gather the `Delete`, `Insert`, `Move`, and `Update` actions performed on the source code AST nodes when SATD is being addressed. Specifically, we compute the *actual* AST edit actions, *i.e.,* those obtained by differencing the `input` and the `target` (*i.e.,* ground truth) of the model.

Subsequently, we create two separate buckets. The first bucket includes all methods where the best-performing model accurately addresses the SATD comment, while the second bucket includes methods for which the suggested code is inconsistent with the developer's proposed repayment (*i.e.,* ground truth). For both categories, we present the relative counts of AST edit actions learned by the model (when the code suggested for addressing the SATD is actually correct) and those where the model faces difficulties in providing a significant implementation.

Also, we perform statistical tests to determine whether one of the experimented techniques is more effective in producing code changes to address SATD. We use McNemar's test [McN47] (with is a proportion test for dependent samples) and Odds Ratios (ORs) on the EMs that the techniques generate. We also statistically compare the distribution of the CrystalBLEU scores (computed at the sentence level) for the predictions generated by each technique by using the Wilcoxon signed-rank test [Wil45]. The Cliff's Delta (d) is used as effect size [GK05] and it is considered: negligible for $|d|$ 0.10, small for $0.10 \leq |d| < 0.33$, medium for $0.33 \leq |d| < 0.474$, and large for $|d| \geq 0.474$. For all tests, we assume a significance level of 95% and we account for multiple tests by adjusting p-values using Holm's correction procedure [Hol79].

Finally, we discuss examples of successfully addressed SATD comments by the top-performing model and, at the same time, we present cases where the model was unable to pay back the TD.

A.4 Results

Table A.2 reports the results obtained by the studied techniques when addressing SATD instances from our test set. The first column ("Model") provides a unique identifier we assigned to each of the 7 experimented techniques described in Section A.3. The "Self-supervised PT" and "Supervised PT" indicates whether a specific configuration we tested featured the two types of pre-training, where the self-supervised is the one adopting the *masked language model* objective and the supervised uses the *generic code changes* dataset to provide the model

Table A.2. Exact Match (*i.e.,* the recommended code is equal to the oracle) and CrystalBleu scores achieved by the different techniques when addressing SATD comments. In bold we report the highest value for both metrics when producing K=1, K=3, K=5, and K=10 candidate removals.

Model	Training Configuration				Top-1		Top-3		Top-5		Top-10	
	Self-supervised PT	Supervised PT	SATD Comm.	FT	EM	CB	EM	CB	EM	CB	EM	CB
M0	✗	✗	✓	✓	0%	22.13%	0%	25.98%	0.0%	25.43%	0.0%	25.14%
M1	✓	✗	✓	✓	2.23%	73.11%	5.40%	73.95%	6.10%	74.19%	7.20%	73.87%
$M2_{CC}$	✓	✓	✓	✓	**2.30%**	**73.41%**	**5.60%**	**74.20%**	**6.70%**	**74.28%**	**8.10%**	**74.25%**
$M3_{CC-Ablation}$	✓	✓	✗	✓	0.9%	72.58%	2.90%	73.48%	3.80%	73.60%	5.10%	73.50%
$M4_{T1}$	✓	✗	✓	✗	1.18%	37.30%	-	-	-	-	-	-
$M4_{T2}$	✓	✗	✓	✗	1.19%	49.68%	-	-	-	-	-	-
$M4_{T3}$	✓	✗	✓	✗	0.0%	1.70%	-	-	-	-	-	-

Table A.3. Comparison among different techniques for top-1 predictions: McNemar's and Wilcoxon's test results.

Comparison	McNemar's Test		Wilcoxon's Test	
	p-value	OR	*p*-value	d
M1 vs. M0	-	-	<0.05	-0.86 (L)
M1 vs. $M2_{CC}$	>0.05	1.0	>0.05	-0.01 (N)
$M3_{CC-Ablation}$ vs. $M2_{CC}$	<0.05	8.0	<0.05	-0.01 (N)
$M4_{T1}$ vs. M2	>0.05	1.38	<0.05	-0.54 (L)
$M4_{T2}$ vs. M2	>0.05	1.36	<0.05	-0.43 (S)
$M4_{T3}$ vs. M2	-	-	<0.05	-0.98 (L)

with knowledge about changing code. The "SATD Comm." column indicates whether the fine-tuning (or the prompting in the case of ChatGPT) included the SATD comment in the model's input, while the "FT" column shows which model has been fine-tuned on our "SATD removal" dataset. Lastly, EM and CB indicate the performance of a specific configuration in terms of (i) the percentage of predictions that are Exact Matches (EM), and (ii) the average CrystalBLEU score (CB) [EP22] for all predictions in the test set. We present the results for different beam sizes (K) of 1, 3, 5, and 10.

Table A.3 reports the results of the statistical tests (McNemar's test and Wilcoxon signed-rank test), with adjusted p-values, OR, and Cliff's d effect size. An $OR > 1$, or a positive Cliff's d indicate that the right-side treatment outperforms the left-side one. To enhance readability, when doing the comparisons, we arranged the treatments to display ORs ≥ 1.

A.4.1 RQ_1: To what extent do pre-trained models of code support the automated SATD repayment?

The first two rows of Table A.2 report the outcomes of the non-pre-trained model (M0) and its counterpart using self-supervised per-training on code and technical language (M1). While M0 is unable to address any SATD (EM = 0.0%) for all values of K, the prediction performances of M1 range from 2.23% (K=1) to 7.20% (K=10). M1 is a CodeT5 fine-tuned for SATD removal and our results stress the importance (and validity) of the pre-training procedure performed on it by the original authors [WWJH21]. Also, the difference in CrystalBLEU with respect to the non-pre-trained model (M0) is statistically significant (p-value < 0.05), according to Wilcoxon signed-rank test, and is accompanied by a *Large* Cliff's

Delta. In the absence of EMs for M0, McNemar's test results cannot be computed.

Answer to RQ_1. The use of a self-supervised pre-trained model (CodeT5) has a significantly positive benefit when addressing SATD, if compared to a non-pre-trained model. The latter is unable to produce exact matches, likely due to the limited size of the fine-tuning dataset.

A.4.2 RQ_2: To what extent does the infusion of "similar-task knowledge" in pre-trained models of code benefits the automated SATD repayment?

Instilling task-similar (*i.e.,* code changes) knowledge into the model ($M2_{CC}$, featuring both self-supervised and supervised pre-training) results in a slight performance improvement as compared to M1 (*i.e.,* self-supervised pre-training only). While there is an improvement across all beam sizes (see Table A.2), this is usually minor. For example, when only relying on the top prediction (*i.e.,* K=1), the EMs rise from 2.23% to 2.30% which, given the 1,000 instances featured in our test set, means 7 new EM predictions. The improvement is slightly higher when looking at higher values of K, with a +0.9% reached for K=10 (7.20% vs. 8.10%). Upon statistically comparing both models (M1 vs. $M2_{CC}$), the McNemar's test (Table A.3) reports a lack of significant differences (p-value > 0.05) in EM predictions between M1 and $M2_{CC}$. Wilcoxon signed-rank test also suggests non-significant differences in the distributions of CrystalBLEU scores between M1 and $M2_{CC}$. Such a result is in line with what was observed by Tufano *et al.* [TPB23].

They found that adopting a pre-training objective resembling the downstream task does not always substantially help, questioning the effort needed for the additional training time. This also seems to be the case when addressing SATD. Despite this, $M2_{CC}$ still is the best-performing model we experimented with and, for this reason, we performed some additional analyses on its predictions.

The EM predictions generated by $M2_{CC}$ with K=10, feature a total of 768 AST *edit actions* correctly implemented by the model. Out of these, 62.36% are `Delete` operations (*i.e.,* an AST node is removed), 33.60% are `Inserts` (*i.e.,* a new node is introduced into the AST), and 2.34% and 1.70% are `Move` and an `Update` operations, respectively  .

When looking at the failure cases (*i.e.,* non-EMs), the distribution of AST *edit actions* that was needed to address the SATD (but that the model failed to reproduce) we found that out of the EMs: 32.01% are `Deletions`, 54.10% `Insertions`, 6.84% `Moves`, and 7.05% `Updates`  . Thus, are not the "types" of AST edits needed to address the SATD that discriminate what the model can or cannot do. Given this finding, we also computed the sheer number of AST *edit actions* that were needed in EMs and wrong predictions to address the SATD. The achieved results, as expected, indicate that the model struggles in addressing SATD in need of a high number of AST changes to be repaid. Indeed, the median number of changes that were required to address the SATD instances that resulted in EMs is 6, as compared to the 20 of the wrong predictions (mean 9.8 *vs* 38.1).

Fig. A.3 illustrates four instances from our test set, including two for which $M2_{CC}$ was able to successfully address the SATD (❶ and ❷), and two for which it failed to pay back the

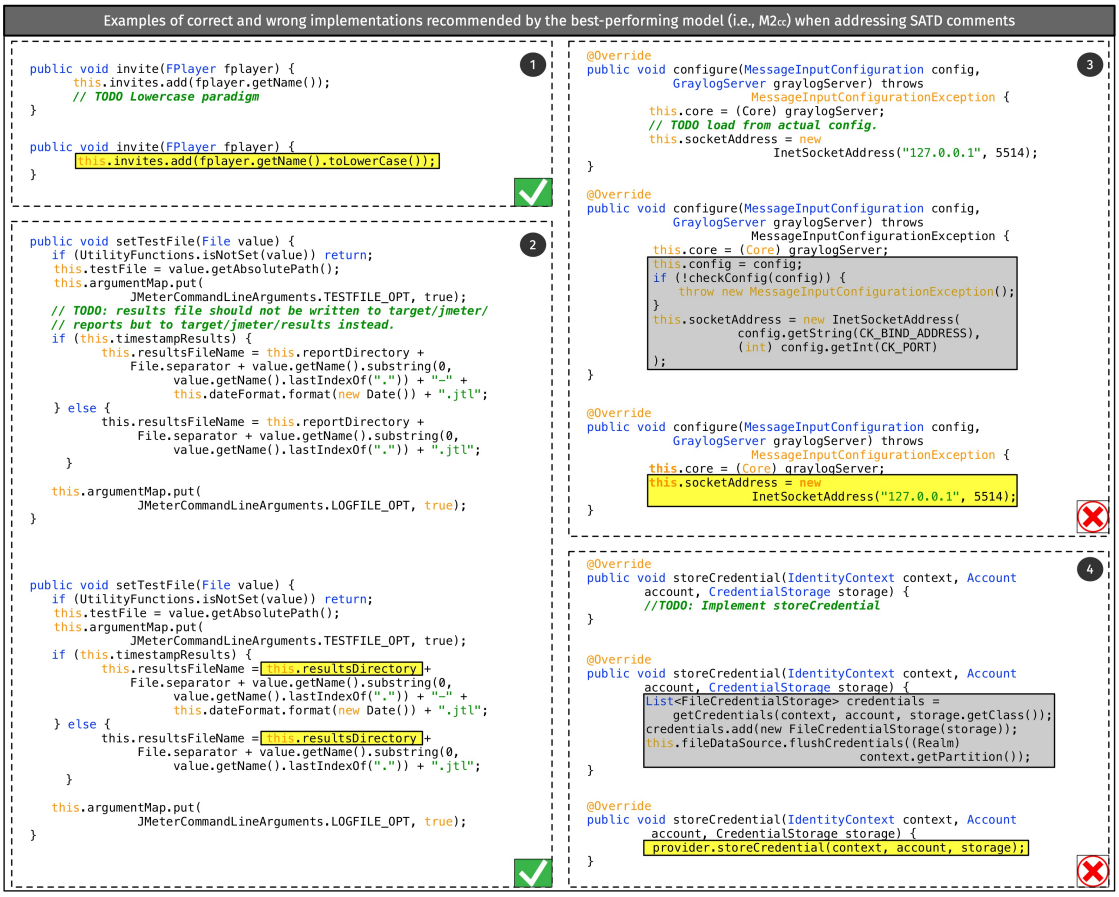


Figure A.3. Example of 4 SATD comments addressed by the model (2 correct and 2 wrong) for top-10 candidate recommendations.

TD (❸ and ❹). For the successful cases, the code on top shows the input method including the SATD, while the one at the bottom shows how the model addressed the SATD (changes highlighted by the yellow boxes). For the failure cases, we also report the expected target from our dataset, namely the code showing how the developers actually addressed the SATD (changes highlighted with grey boxes).

In ❶ the SATD mentions "`TODO Lowercase paradigm`". To fulfill this requirement, the model performs two distinct AST edit actions, addressing the TD in the right location by injecting the `toLowerCase` invocation. The example of scenario ❷ shows how the model addresses the SATD by replacing the *reportDirectory* attribute in the current instance with *resultsDirectory* through a code change that involves updating two AST nodes (*i.e.,* an `Update` edit) in the *if* and *else* statements.

In scenario ❸ the model fails to effectively handle the SATD comment (`TODO load from actual config`) by requiring modifications to the instantiation of `this.socketAddress`. The hard-coded *IP address* and *port* need to be replaced with values fetched using the `config` method. The implementation of these changes is unsuccessful, as the model outputs the same

method (M_{satd}) with the SATD comment removed (gray box in ❸). There are a total of 21 AST edit actions to be implemented to successfully address the SATD comment, including 18 `Insert` operations, two `Delete` operations, and one `Update` action. When considering scenario ❹, the SATD left by the developer requires the complete implementation of the *storeCredential* method. Nonetheless, the recommendation provided by the model does not address the SATD comment appropriately since it assumes the existence of a *storeCredential* method that takes *context, account*, and *storage* as input parameters, failing to address the TD. A successful change would have required the addition of 19 new nodes (*i.e.,* `Insert`) to the AST of the *Java* method *storeCredential*.

Answer to RQ$_2$. Seeding task-similar knowledge (*e.g.,* code changes) into a model pre-trained using self-supervised task models of code only slightly improves their performance. Even our best-performing model struggles in addressing SATD instances requiring a large number of AST edit actions.

A.4.3 RQ$_3$: To what extent does the presence of "context-specific knowledge" help pre-trained models of code in the automated SATD repayment?

By comparing the results in row 4 of Table A.2 ($M_{CC-Ablation}$) with those in row 3 (M_{CC}), we can observe the fundamental role played by the context-specific knowledge provided as input to the model (*i.e.,* the SATD comment) in automatically addressing TD. Admitting TD through a comment aids the model to better perform for all considered beam sizes (K). For example, when focusing on a single candidate solution (*i.e.,* top-1), $M3_{CC-Ablation}$ can only achieve an EM in 0.9% of cases, while $M2_{CC}$ does it in 2.30% of cases. When looking at higher values of K, the gap becomes even larger, up to a +3% for $K = 10$ (5.10% vs. 8.10%). McNemar's test indicates a significant difference (p-value < 0.05) between $M2_{CC}$ and $M3_{CC-Ablation}$, with $M2_{CC}$ having 8 times higher odds (OR=8) to address SATD than $M3_{CC-Ablation}$. Instead, although the differences found by Wilcoxon signed-rank test for the CrystalBLEU are statistically significant, the effect size is *negligible*. The obtained results further highlight the importance for developers to admit TD. In essence, SATD does not only serve as a trace for themselves and for other developers [ZFSD21], but, also, as a way to better guide automated tools in addressing TD. Furthermore, this stresses the importance of recommending developers that TD should be admitted [ZNA+17].

Answer to RQ$_3$. The availability of context-specific knowledge in the form of SATD comments enhances the performance of pre-trained models of code, allowing them to achieve a substantial increase in the percentage of automatically addressed TD. This is a further motivation for developers to admit TD in their source code.

A.4.4 RQ_4: Are general-purpose large language models zero-shot learners for SATD repayment?

The last three rows of Table A.2 report the performances achieved by ChatGPT as *zero-shot learner* for addressing SATD. Two findings emerge: (i) the usage of different templates to prompt ChatGPT for the task of SATD repayment plays a crucial role and, (ii) the performances achieved when recommending one single candidate solution (top-1) are lower than DL-based techniques appositely fine-tuned (M1 and $M2_{CC}$) to pay back TD. As for the use of several prompt templates, it is important to note that designing templates showing the code first and the comment later (as did in $M4_{T3}$) strongly penalize the model, resulting in 0 EM and the lowest CrystalBLEU score across all treatments. Differently, providing the SATD comment first and the code including such a comment later helps ChatGPT in achieving better performances, with 1.18% and 1.19% of SATD comments successfully addressed, respectively for $M4_{T1}$ and $M4_{T2}$.

McNemar's test on the top-1 recommendation indicates no significant differences (p-value > 0.05) between $M4_{T1}$ and $M2_{CC}$, as well as $M4_{T2}$ and $M2_{CC}$. However, there are statistically significant differences when comparing the CrystalBLEU distributions of the tested templates with $M2_{CC}$. In these instances, the effect is *large* for $M4_{T1}$ and $M4_{T3}$, and small for $M4_{T2}$, where ChatGPT performed best.

Without knowing the details of ChatGPT's implementation, it is hard to speculate on the reasons behind such performances. Possibly, they could be related to the lack of a specific fine-tuning for the specific task, or also to the need to generate suitable prompts.

Answer to RQ_4. When used in a zero-shot setting, LLMs, and ChatGPT in particular, exhibit sub-optimal performance compared to pre-trained models of code appositely tuned for the specific task of addressing SATD. Additionally, the use of well-crafted templates plays a crucial role for LLMs being used off-the-shelf.

A.5 Threats to Validity

Construct validity. The main issue to be considered is whether the observed SATD removals are true positives. To mitigate this threat, we only considered cases in which the SATD comment contained well-recognized keywords, and we manually analyzed a sample to check for problematic cases.

To avoid training our model on instances where the SATD was removed by chance, we excluded cases where the affected method was removed. We are, however, aware that the latter heuristics, while mitigating some threats to construct validity, could affect the study's generalizability.

Internal validity. As explained in Section A.2, we used the default settings of the employed language models. Better results could be achieved with proper hyperparameter tuning. The assumption we have made in Section III-B which links the removal of an SATD comment with the change performed within the same method is subject to imprecision. First,

the commit could tangle the SATD repayment with other changes. Second, the comment removal may be out of sync with the source code change aimed at addressing the SATD.

Conclusion validity. The comparison between different techniques is supported by suitable statistical procedures and effect size measures. Also, the results of multiple tests have been adjusted through Holm's procedure [Hol79]. We are aware that the performance of the studied approaches may change, and possibly improve, if experimenting with a larger dataset.

External validity. Our study is limited in terms of programming language (Java) and, more important, the SATD removal dataset is based on 1,000 instances. As stated in Section A.2, we limit to addressing SATD that have been resolved within the same method, and to methods not longer than 1024 tokens. As the paper aims to set—within the employed generative models—an "upper bound" of the SATD repayment capability, we do not expect any better results for more complex and extensive changes. Better generalizability of our results would require studies on further and more diversified datasets.

In terms of considered models, our results are limited to the CodeT5-base [WWJH21], as well as a zero-shot attempt done with ChatGPT [cha]. Other models having a different size and architecture could, possibly, exhibit different results. However, in this circumstance, our interest was to mainly show the feasibility of SATD removal and, within the same model (CodeT5 in our case), the relative improvements with different levels of pre-training and fine-tuning. Moreover, we acknowledge that we have not experimented with edit-specific models, such as CoditT5 [ZPN+22b], and therefore further experimentation with such models would be desirable. At least, we partially mitigated this threat by experimenting in RQ_2 a pre-training with code changes. Moreover, as explained in Section II-C, our dataset mostly features removals and additions rather than updates and moves.

Last, but not least, also the preliminary results with ChatGPT need to be confirmed or confuted with similar yet differently implemented tools, *e.g.,* Google Bard [Goo23].

A.6 Conclusions and Future Work

In this paper, we investigated the use of Deep Learning (DL) models for automatically addressing technical debt (TD). To train the models, we leveraged 5,039 instances of SATD removals mined using an existing tool [ACB+22]. Such a small number of training instances made the pre-training of the model absolutely necessary to achieve an automated fixing of SATD instances. Nevertheless, even the best-performing model we experimented with can automatically repay SATD only in a minority of cases (2% to 8%). The complexity of such a task has also been confirmed by the results achieved exploiting the state-of-the-art LLM (*i.e.,* ChatGPT [cha]), that under-performed if compared to the specialized models we tested. So far, the use of generative deep learning models has been successful for several code tasks that require either generation of new code, or the change of existing one. Some (non-exhaustive) examples of achieved performances are on the order of ∼14% for bug fixing (Wang *et al.* [WWJH21]), of ∼5% for generating a reviewed version of existing source code (Tufano *et al.* [TMM+22]), and ∼23% for generating code blocks (Ciniselli *et al.* [CCP+21]). As far as SATD repayment is concerned, we have observed both positive results and negative results.

On the positive side, results are in line with some existing approaches for automated bug fixing [TWB+19a].

On the negative side:

1. The level of performances achieved so far would still limit the applicability of the proposed approach in real practice. The latter may also possibly require some explanation/rationale of the changes to be performed, *e.g.,* telling to the developer that the change being done is enacting a refactoring action, fixing a bug, improving the code readability, etc.
2. Such results have been obtained under the assumption that the change concern a single method, with a limited maximum size of the considered methods, and assuming that the removal of a SATD comment would correspond to changes aimed at addressing it.
3. Although the proposed approach was able to correctly recommend instances of SATD repayment involving the addition of an AST (over 33% of our exact matches), the approach may fail to introduce a totally new, unseen piece of feature, as well as it is unlikely to be able to perform changes such as API upgrade/replacements.

The aforementioned limitations greatly stimulate future research in this area. First and foremost, although in RQ_2 we have performed a large training of CodeT5 on a dataset of code changes, it would be worthwhile to experiment with models more specifically suited for code edits, such as CoditT5 [ZPN+22b].

Second, although RQ_4 has shown that a zero-shot instance of ChatGPT fails to support the SATD repayment task, other investigations with LLMs are worthwhile, as those might be able to help in supporting larger change tasks. They can go in the direction of prompt engineering, as well as experimenting with other LLMs. Third, given the variety of the SATD nature, it may be worthwhile to pursue the development of eclectic approaches, that first classify the type of needed change (*e.g.,* along the line of what SARDELE [ZSD20] does) and then employing different models or even different approaches for each of them. Our future work targets (i) the possibility to exploit LLMs in a few-shot learning scenario (rather than the zero-shot we experimented with), and (ii) the definition of different mining pipelines which could help enlarging the SATD removal dataset, possibly boosting performance.

B

Toward Automatically Completing GitHub Workflows

Setting and maintaining a continuous integration and delivery (CI/CD) pipeline is crucial for any software project. Indeed, CI/CD contributes to enhancing software quality and developers' productivity [Che17], and to speed up release cycles [VYW+15]. Nevertheless, previous research has highlighted the challenges encountered by developers in setting up and maintaining CI/CD pipelines [Che15, HNT+17, ZVP+20, ZNDP22, SN23]. Despite the availability of modern CI/CD infrastructures and reusable assets (*e.g.,* GitHub actions), the intrinsic CI/CD requirements and underlying technology of a given project may still make this task hard [HNT+17, ZNDP22]. For example, this could be the case when a system needs to be deployed and tested on different operating systems or even embedded devices.

The aforementioned challenges entail the need for recommender systems helping developers in setting up and maintaining CI/CD pipelines. This is also supported by a study by Soroar *et al.* [SN23], reporting that ∼60% of the 90 developers they surveyed encountered difficulties in automating workflows using GitHub actions.

It is worth mentioning that the possible solutions are somewhat similar to those related to automated code completion, where approaches have been defined either to provide suggestions about non-trivial, generic code elements (up to blocks) to be completed [CCP+21], or more specialized suggestions, *e.g.,* related to creating assertions [WTM+20], or repairing vulnerabilities [CKM22, FTL+22b] and bugs [CKT+19, LWN20, LWN22b].

That being said, helping developers in setting up a CI/CD pipeline poses unique challenges. Indeed, the structure a CI/CD pipeline mixes up very specific scripting elements (*e.g.,* related to configuring a server, downloading certain libraries, etc.) with some more recurring and regular reusable elements (*e.g.,* the actions in the case of GitHub), up to natural language elements. Also, CI/CD pipeline contain several extremely context-specific elements, such as paths of installation directories, or URLs of resources to download. This creates major challenges to the use of data-driven techniques for the automated recommendations of these elements.

This paper proposes GH-WCOM (GitHub Workflow COMpletion) an approach leveraging Transformer models [VSP+17] to provide automated completion of GitHub workflows. To

develop (and train) GH-WCOM, we have leveraged the existing body of GitHub workflows starting from a dataset by Decan *et al.* [DMMG22].

To make a GitHub workflow completion possible, we have defined and implemented a multi-step pre-processing including an abstraction of the tokens for which their verbatim prediction would not be feasible (*e.g.,* a very specific path in a project) while still leaving to GH-WCOM the ability to recommend some very peculiar workflow elements such as tool options and other scripting elements. GH-WCOM can recommend GitHub workflow completions in different modes that mimic how a developer may implement the workflow, *i.e.,* (i) suggesting the next statement (a GitHub step), or (ii) helping to complete a job with implementation elements once the developer has defined, in plain English, what the job should do.

Summarizing, this paper makes the following contributions:

1. We propose GH-WCOM, which, to the best of our knowledge, is the first approach to automatically complete CI/CD pipelines, and GitHub workflows in particular.
2. We experiment with different pre-trainings, abstraction levels, and completion scenarios. Results indicate that pre-training at least on English text is required, and GH-WCOM's performance for correct prediction is ∼34%. The correct prediction accuracy is correlated with the model's confidence.
3. We report a qualitative analysis discussing the extent to which the recommendations provided by GH-WCOM could still be helpful also when the generated output is different from the target (expected) one. Also, we discuss how GH-WCOM is competitive with respect to recent, popular general-purpose recommenders based on large language models, *e.g.,* CoPilot [cop] and ChatGPT [cha].
4. We made publicly available GH-WCOM scripts, checkpoints predictions, and the used datasets [repc].

B.1 Background

GitHub workflows integrate CI/CD in the GitHub infrastructure. A GitHub workflow (example in the top part of Fig. B.1, while the bottom part will be described later in the paper) is a YAML file located under the `.github/workflows` (sub)directory of a repository. As specified by the `on:` clause, a workflow is triggered based on some events (*e.g.,* a push, a pull request) and executes a series of jobs, specified after the `jobs` keyword (as the job named `build` in the figure).

Jobs are units of execution of a CI/CD process and can run in parallel or sequentially (if dependencies between jobs are specified) on runners. Unless they use explicit ways to exchange information (*e.g.,* uploading and downloading artifacts in a storage area), jobs are independent of each other. Runners can be local or remote virtual machines or containers. Runners and containers are specified after the job name, using the `runs-on` clause, and, if containers are used, the `container:` and `image` clauses. The job in the example runs on an Ubuntu virtual machine and uses a container from an image bringing the *gcc* compiler. Each job consists of a sequence of steps. In Fig. B.1, steps are all items preceded by a dash following the keyword `steps`. There are two ways to implement a step. The first (denoted

Example of GitHub Workflow

YAML Representation

```yaml
name: CBuild

on:
  push:
     branches: [ main ]
  pull_request:
     branches: [ main ]

jobs:
  build:
    runs-on: ubuntu-latest
    container:
      image: gcc
  steps:
   - name: checking out the repository
     uses: actions/checkout@v2
   - name: Running makefile to compile the program
     run:  make
```

JSON-like Representation

```
{
  "name": "CBuild",
  "on": {
    "push": {
      "branches": [
        "main"
      ]
    },
    "pull_request": {
      "branches": [
        "main"
      ]
    }
  },
  "jobs": {
    "build": {
      "runs-on": "ubuntu-latest",
      "container": {
        "image": "gcc"
      }
    },
    "steps": [
      {
        "name": "checking out the repository",
        "uses": "actions/checkout@v2"
      }
    …
  }
}
```

Figure B.1. GitHub workflow example

by the keyword `uses`) is to leverage GitHub actions, *i.e.,* reusable applications available on GitHub that implement recurring tasks. For example, the `actions/checkout@v2` is version 2 of an action checking the content of the GitHub repository branch on which the workflow has been triggered.

The second (keyword `run`) consists of directly executing whatever application is available in the virtual machine/container (*e.g.,* `apt-get` to install components, `gradle` to run a Gradle build). Run steps are typically used for specific tasks for which an action is not available, or the task is so simple as to not require an action. Optionally, a step can be documented with a textual description of its action or run command, using the `name` keyword. Further information about GitHub workflows and actions is available on the GitHub documentation [gita].

B.2 GH-WCOM

This section describes GH-WCOM, the proposed approach to recommend GitHub workflow completions. GH-WCOM leverages the Text-to-Text Transfer Transformer (T5) model by Raffel *et al.* [RSR+20]. First, we pre-train T5 by experimenting with different strategies. Then, we train the tokenizer needed by GH-WCOM and, after an hyperparameter calibration, we fine-tune T5 with instances specifically related to the actual prediction tasks. After that, we use the trained model for two different kinds of predictions, *i.e.,* (i) adding the next step in a workflow job, or (ii) completing a job whose steps have just been specified in terms of natural language text.

In the following, after overviewing the T5 model, we describe the different steps of the approach.

B.2.1 An overview of T5

T5 [RSR+20] is an encoder-decoder Transformer [VSP+17] designed to work in a text-to-text setting. Whatever the generation task is, T5 can be employed as long as both the input and the output can be represented as textual strings (*e.g.,* translating from English to Spanish, outputting the fixed version of a provided buggy code). We have chosen T5 given its successful application in several code completion/generation tasks [MPB22, CCP+21, TMM+22, WWJH21].

The training procedure of T5 is usually performed in two steps. First, the model is pre-trained on a large-scale dataset using self-supervised training. The pre-training provides T5 with general knowledge about the language(s) of interest. For example, assuming the will of building an English-to-Spanish translator, we could provide as an input to the model English and Spanish sentences having 15% of their tokens masked, with the model in charge of predicting them. That makes the pre-training fully self-supervised.

Subsequently, the model undergoes fine-tuning, which is supervised training (*e.g.,* providing pairs composed of an English sentence and its Spanish translation). Fine-tuning specializes the model for the task of interest.

Raffel *et al.* experimented with five T5 variants, differing in terms of the number of trainable parameters: small, base, large, 3 billion, and 11 billion. Considering our computational resources and recent successful application of $T5_{small}$ to automate code-related tasks [MPB22, CCP+21, TMM+22, WWJH21], we opted for the simplest architecture which still features 60M trainable parameters, consistently with large language models used in the literature. For additional architectural details, we point the reader to the work by Raffel *et al.* [RSR+20].

B.2.2 Abstraction

We conjecture (and will later experiment) that learning to autocomplete GitHub workflows on raw text (*i.e.,* with no preprocessing) is extremely challenging. This is mainly due to the presence of context-specific (and often unique, *i.e.,* they have not been seen before) elements in the workflows, such as `paths` and `urls`. For example, the left part of Fig. B.2

shows a GitHub workflow featuring elements such as the `./vendor/bin/phpunit` path or the specific version of an action the user is using (*e.g., actions/checkout@v2*), which are likely to hinder the completion learning. These are some of the elements we aim at abstracting with special tokens (*e.g.,* replacing a path with the <PATH> tag), as it can be seen in the right part of Fig. B.2.

Such an abstraction moves the definition of these context-specific elements from T5 (now only in charge of indicating the need for *e.g.,* a <PATH>) to the developer. We acknowledge that this might imply a slightly higher effort on the developer's side who needs to "fill the placeholders" (*i.e.,* the special tags) in the prediction.

To define the abstraction rules, we leverage the unique set of tokens extracted from the workflows of the projects listed in the GitHub actions dataset by Decan *et al.* [DMMG22]. The dataset features 67,870 GitHub repositories, 29,778 of which use GitHub workflows, and is the one we use to create our training and testing datasets as described in Section B.2.3. Given the list in that dataset, we were able to clone 69,040 GitHub repositories, which is more than the 67,870 for which Decan *et al.* extracted workflow data. From those, we retrieved all GitHub workflows and extracted their "tokens". A token can be an action name, a command to run, the option of a command, a path, etc. Out of 10,188,342 unique tokens, 284,463 appear in one workflow, *i.e.,* are very specific, confirming our conjecture about the need for abstraction. We randomly selected 1,000 of those tokens for manual inspection. We clustered them based on their "type" (*e.g.,* `path`, `file`). Such a process has been performed by the first author, with the results checked by three other authors. Such a process led to the definition of five categories of context-specific tokens we aim at abstracting: `url` (*i.e.,* a reference to a web resource, such as an IP address), `file` (*i.e.,* a file name/path), `path` (*i.e.,* a path to a directory or to any other resource which cannot be identified as a file since lacking extension), `version number`, (*i.e.,* the specific version of a library, language, or other resources being used), and `action version` (*i.e.,* the specific version of an action that is used). For each category, we defined a special token acting as a placeholder during the abstraction. Note that we distinguish between `version number` and `action version` since we assume this could provide additional information to the model which might be useful for the learning.

The abstraction example reported in Fig. B.2 shows how we replace the `action version` of the token `actions/checkout@v2` with the special <PLH> token, while `files` and `urls` such as `bin/install-wp-test.sh` and `127.0.0.1` are replaced with ⟨FILE⟩ and ⟨URL⟩, respectively. The code implementing our abstraction process is publicly available [repd]. In a nutshell, we use regular expressions and heuristics to identify the token types of interest and abstract them. The identification of `files` leverages, besides a regular expression, a list of extensions we defined during the manual analysis of the tokens appearing in a single workflow. Such a list is also provided in our replication package [repd].

To validate our choice of the specific tokens to abstract, we extracted all single-occurring tokens in our dataset, namely those certainly representing problematic cases for any data-driven technique. In total, we identified 23,273 distinct single-occurring tokens. Out of these: 8,226 (37%) are `paths`, 8,068 (35%) are `files`, 2,833 (12%) are `urls`, and 2,334 (10%) are `versions`. This means that ∼93% of single-occurring tokens are abstracted by

our procedure. This indicates that the proposed abstraction strategy is suitable to abstract rarely-occurring tokens.

Workflow Raw Tokens

```
name: PHPUnit

on:
  push:
    branches:
      - develop
      - trunk
    paths:
      - '**.php'
  pull_request:
    branches:
      - develop

jobs:
  phpunit:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout
      uses: actions/checkout@v2

    - uses: getong/mariadb-action@v1.1

    - name: Set PHP version
      uses: shivammathur/setup-php@v2
      with:
        php-version: '7.4'
        coverage: none
        tools: composer:v1

    - name: Install dependencies
      run: composer install

    - name: Setup WP Tests
      run: bash bin/install-wp-tests.sh
           wordpress_test root '' 127.0.0.1

    - name: PHPUnit
      run: './vendor/bin/phpunit'
```

Workflow Abstracted Tokens

```
name: PHPUnit

on:
  push:
    branches:
      - develop
      - trunk
    paths:
      - '<FILE>'
  pull_request:
    branches:
      - develop

jobs:
  phpunit:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout
      uses: actions/checkout<PLH>

    - uses: getong/mariadb-action<PLH>

    - name: Set PHP version
      uses: shivammathur/setup-php<PLH>
      with:
        php-version: '<V_NUMBER>'
        coverage: none
        tools: composer:v1

    - name: Install dependencies
      run: composer install

    - name: Setup WP Tests
      run: bash <FILE> wordpress_test
           root '' <URL>

    - name: PHPUnit
      run: '<PATH>'
```

Figure B.2. Example of Raw and Abstracted Instance.

B.2.3 Training and Testing Datasets

B.2.3.1 Pre-training dataset

Since the goal of pre-training is to provide T5 with general knowledge about the language(s) of interest, we built a pre-training dataset featuring `YAML` files (*i.e.,* the language used in GitHub workflows), and in particular both general-purpose `YAML` files as well as those implementing GitHub actions. The former are used for various purposes, *e.g.,* CI/CD scripts for other infrastructures (*e.g.,* Travis-CI) or other configuration files.

GitHub actions feature a syntax closer to workflows and therefore would provide further knowledge during pre-training.

We collected general-purpose `YAML` files in two steps. First, we searched for `YAML` files in the 69,040 GitHub repositories we cloned, while excluding those implementing GitHub workflows that we will use to fine-tune the model (*i.e.,* those contained in the `./github/workflows/` directory). This resulted in 443,037 general-purpose `YAML` files.

To further expand this dataset, we cloned all public non-forked repositories having at least 100 stars and 100 commits, and created in the time range that goes from 2022-25-01 (*i.e.,* the day after Decan *et al.* built their dataset) to 2022-30-09 (the day in which we performed the data collection). The identification of these repositories has been performed using the GitHub search platform by Dabić *et al.* [DAB21].

We successfully cloned additional 1,124 GitHub repositories that are not in the dataset by Decan *et al.* nor are forks of those. To create the pre-training dataset, which counts a body of 146,006 general-purpose `YAML` files, we excluded duplicated instances as well as those including non-ASCII tokens and all those having $\#tokens \geq 1024$. Fixing an upper-bound in terms of the number of tokens for the model's input helps in taming the computational cost of training and is a common practice in the literature exploiting DL models to automate code-related tasks [HLWM20, WXL+21, MAPB21, TPT+21, MAPB22, CCP+21].

Concerning the `YAML` files implementing GitHub actions, we collected 13,638 unique examples about the usage of actions from the GitHub Marketplace [mar].

The pre-training dataset features 146,066 general-purpose `YAML` files and 13,638 `YAML` files implementing GitHub actions. Each instance in the dataset is a pair featuring (i) a `YAML` file with 15% of its tokens randomly masked, and (ii) the expected target, namely the tokens the model is expected to predict instead of the masked ones.

B.2.3.2 Fine-tuning dataset

Our fine-tuning dataset features 73,708 GitHub workflows from the whole body of GitHub projects made available by Decan *et al.* [DMMG22]. On top of those, we mined 733 workflows from the 1,124 GitHub repositories previously mentioned.

We removed duplicated workflows, and, as done before, all those having $\#tokens \geq 1024$, instances containing non-ASCII characters, and those which overlap with instances in the pre-training dataset. We were left with 17,935 unique workflows used to train and evaluate GH-WCOM. These workflows feature an average of 54 lines (median=41) and 120 tokens (median=84).

We split the dataset into training (80%), validation (10%), and test (10%), making sure that all the instances coming from the same project are assigned to the same subset, thus avoiding leakage of data among the three sets. We obtained 14,348 workflows to train the models, 1,793 for hyperparamenter tuning, and 1,794 to test the best configuration identified. Each workflow is represented as a JSON-like object preserving the structure of the original workflow file, as it can be seen in the bottom part of Fig. B.1.

We then fine-tune GH-WCOM to support two workflow completion scenarios. In the first one, *next step* (NS_{task}), GH-WCOM is in charge of predicting the complete n^{th} step a developer is likely to write in a workflow given the preceding already written tokens. A step may or may not contain a textual description (*name*), and it can either consist of action invocations (*uses*) or commands (*run*). In the second scenario, job completion (JC_{task}), GH-WCOM gets as input an abstract job where only *names* are specified, and it is asked to complete it step by step. Fig. B.3 helps in better understanding these two scenarios by depicting a fine-tuning instance from our dataset.

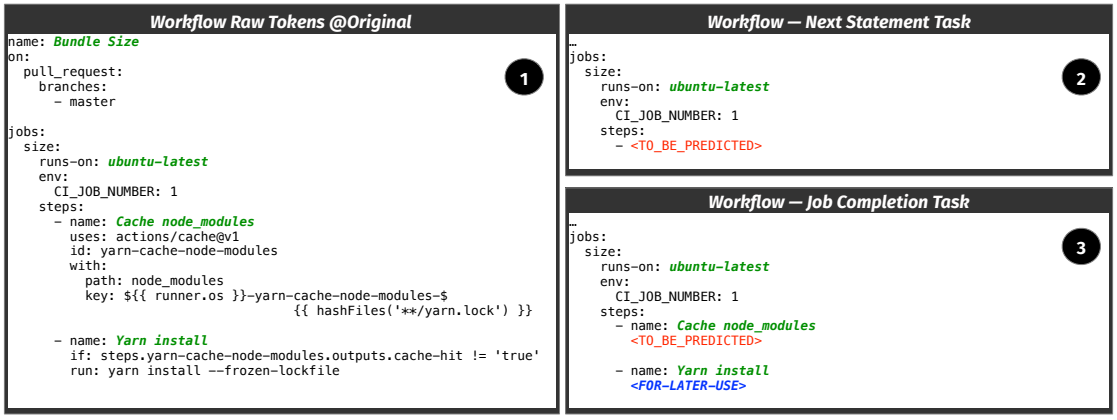


Figure B.3. Example of instance for fine-tuning the T5 model on both tasks, namely NS_{task} and JC_{task}

Since we experiment with both the raw workflow version (*i.e.,* no abstraction) and with its abstracted version, we report in Fig. B.3 an example of "raw instance". The left part of the figure ❶ shows the original GitHub workflow, while ❷ depicts its version for fine-tuning the model for NS_{task} . In this case, we are simulating a scenario in which the developer already wrote the first 11 lines of the workflow (*i.e.,* up to `steps:`), and GH-WCOM is asked to predict the first step of the job (*i.e.,* `uses: actions/checkout@v2`). Note that we can extract multiple (5) training instances from this workflow. Indeed, we can ask the model to predict the first step of the job given just the preceding statements.

Then, we can ask the model to predict the second step also given the definition of the first step, etc. Fig. B.3 ❸ depicts a fine-tuning instance for JC_{task} . In this case, we assume that the developer wrote the skeleton of a job by only defining, when available, the job's name it should feature (*e.g., Yarn install*). The model is in charge to predict the step masked with the `<TO_BE_PREDICTED>` token, while the `<FOR-LATER-USE>` token is used to indicate steps that are not yet implemented. Also in this case we can build multiple fine-tuning instances from the workflow in Fig. B.3. We can start predicting the first step in a job using the following $n-1$ for which only the name is provided; then, we can predict the second step, providing the model with the full implementation of the first (as if the model already predicted it) and the following partially defined $n-2$ as context; etc.

Table B.1 reports the number of instances in the training, validation, and test datasets for both completion scenarios.

Dataset	train	eval	test
Pre-training	159,645	-	-
Fine-tuning: NS_{task}	108,900	13,009	13,630
Fine-tuning: JC_{task}	108,900	13,009	13,630

Table B.1. Number of instances in the used datasets

B.2.4 Training and Hyperparameter Tuning

All the trainings we performed have been run using a Google Colab's 2x2, 8 cores TPU topology with a batch size of 32 and an input and target sequence length of 1,350 and 750 tokens, respectively.

B.2.4.1 Tokenizer Training

Since our task is characterized by the presence of natural language and human-readable data-serialization language (*i.e.,* `YAML` data), we trained a new tokenizer (*i.e.,* a SentencePiece model [Kud18] with vocabulary size set to 32k word-pieces) to cope with context-specific elements. To this extent, we use the 159,645 YAML files included in our pre-training dataset and 712,634 English sentences from the C4 dataset [RSR+20]. The latter is a common practice in literature when developing DL-based models that are required to deal with multi-modal data such as code and technical natural language [MPB22, WWJH21]. We included English sentences due to the presence of technical English occurring within GitHub workflows.

B.2.4.2 Pre-training strategies

We assess GH-WCOM in four pre-training scenarios. The first is *No pre-training* ($T5_{NO-PT}$), in which the model is not pre-trained, but directly fine-tuned. This means that the model has no previous knowledge of any language and it is just trained to complete GitHub workflows with the available fine-tuning dataset composed by ∼109k instances. The second is *YAML pre-training* ($T5_{YL}$), in which the model is first pre-trained for 300k steps on a total of 159,645 `YAML` files including 13,638 actions from the GitHub Marketplace [mar] and then fine-tuned on the workflow completion task. Thus, in this case the model has knowledge of the general structure of `YAML` files before being then specialized on the completion task. The third is the *Natural Language Pre-training* ($T5_{NL}$), for which we fine-tune the publicly available checkpoint by Raffel *et al.* [t5-] which has been pre-trained for 1M steps on English sentences from the C4 dataset [RSR+20].

The fourth scenario is *Natural Language+YAML Pre-training* ($T5_{NL+YL}$) in which we further pre-trained the previously mentioned checkpoint for additional 300k steps on YAML files, reaching a total of 1,3M pre-training steps (1M on English sentences + 300k on YAML files).

B.2.4.3 Hyperparameter Tuning

Once pre-trained the models, we fine-tune the hyperparameters of the model following the same procedure employed by Mastropaolo *et al.* [MSC+21].

In particular, we assessed the performance of T5 when using four different learning rate schedulers: (i) *Constant Learning Rate* (C-LR): the learning rate is fixed during the whole training; (ii) *Inverse Square Root Learning Rate* (ISR-LR): the learning rate decays as the inverse square root of the training step; (iii) *Slanted Triangular Learning Rate [HR18]* (ST-LR):

the learning rate first linearly increases and then linearly decays to the starting learning rate; and (iv) *Polynomial Decay Learning Rate* (PD-LR): the learning rate has a polynomial decay from an initial value to an ending value in the given decay steps. The exact configuration of all the parameters used for each scheduling strategy is reported in our replication package [repd]. Such a procedure has been performed for each of the fine-tuning datasets previously described (*i.e.,* both tasks on raw and abstracted code).

Having four different training scenarios, four possible learning rates, two different completion contexts, and two versions of the fine-tuning dataset (*i.e.,* abstracted and raw tokens), the hyperparameter tuning required building and evaluating 64 models. We fine-tuned each model (*i.e.,* each configuration) for 100k steps. Then, we compute the percentage of correct predictions (*i.e.,* cases in which the model can correctly generate a recommendation) in the evaluation set. Table B.2 reports the achieved results for each of the 64 models we fine-tuned to find the best-performing configuration (which is reported in boldface).

Table B.2. Hyperparameters tuning results

No Pre-training				
	Raw		**Abstracted**	
	NS_{task}	JC_{task}	NS_{task}	JC_{task}
Constant (C-LR)	11.06%	19.24%	13.27%	26.73%
Inverse Square Root (ISQ-LR)	**12.38%**	**21.13%**	**14.21%**	**27.86%**
Slanted Triangular (ST-LR)	10.13%	20.95%	12.81%	26.65%
Polynomial Decay (PD-LR)	10.86%	19.01%	13.78%	25.57%
YAML Pre-training				
	Raw		**Abstracted**	
	NS_{task}	JC_{task}	NS_{task}	JC_{task}
Constant (C-LR)	**16.26%**	25.92%	19.05%	32.35%
Inverse Square Root (ISQ-LR)	15.77%	25.47%	18.93%	31.22%
Slanted Triangular (ST-LR)	14.26%	24.73%	18.05%	30.96%
Polynomial Decay (PD-LR)	16.15%	**26.01%**	**19.24%**	**32.81%**
English Pre-training [RSR$^+$20]				
	Raw		**Abstracted**	
	NS_{task}	JC_{task}	NS_{task}	JC_{task}
Constant (C-LR)	18.35%	27.18%	22.25%	34.02%
Inverse Square Root (ISQ-LR)	18.36%	27.10%	21.70%	33.91%
Slanted Triangular (ST-LR)	17.67%	26.61%	21.70%	33.25%
Polynomial Decay (PD-LR)	**18.46%**	**27.47%**	**22.30%**	**34.12%**
YAML+English Pre-training				
	Raw		**Abstracted**	
	NS_{task}	JC_{task}	NS_{task}	JC_{task}
Constant (C-LR)	18.06%	27.40%	21.55%	32.91%
Inverse Square Root (ISQ-LR)	**18.36%**	**28.17%**	**21.84%**	**34.62%**
Slanted Triangular (ST-LR)	16.50%	25.90%	18.88%	32.11%
Polynomial Decay (PD-LR)	18.28%	27.33%	21.40%	33.36%

B.2.4.4 Fine-tuning

Once identified the best learning rates to use, we fine-tuned the final models using early stopping to avoid overfitting. In particular, we save checkpoints every 10k steps using a delta of 0.01, and a patience of 5. This means training the model on the fine-tuning dataset

and evaluating its performance on the evaluation set every 10k. The training procedure stops if a gain smaller than the delta (0.01) is observed at each 50k step interval and the best-performing checkpoint up to that training step is selected. Complete data about this process is available in our replication package [repd].

B.2.4.5 Generating Predictions

After the model has been trained, we can generate predictions for the task we aim at supporting using different decoding schema. To this end, we opted for a greedy decoding strategy [SVL14] that generates the recommendation, by selecting at each decoding step the token with the highest probability of appearing in a specific position. Thus, a single prediction is generated for an input sequence.

B.3 Study Design

The *goal* of our study is to evaluate GH-WCOM. The *quality focus* is GH-WCOM's ability to provide correct predictions, as well as predictions that, while differing from the ground truth, could still be valuable for developers. We focus on the two completion scenarios previously described: NS_{task} (mimicking a top-down coding adopted by the developer when writing the workflow statement by statement), and (ii) JC_{task} (helping the developer to complete a job with implementation elements given its textual description). The context consists of the test datasets summarized in Table B.1.

The study aims at answering the following research questions:

RQ_1: *How does GH-WCOM perform with different pre-training strategies?* RQ_1 assesses the impact of using different pre-training strategies when completing workflows. We experiment with four pre-training strategies, including the lack of pre-training.

RQ_2: *How does GH-WCOM perform for different prediction scenarios?* RQ_2 tests GH-WCOM in different prediction scenarios, *i.e.,* next statement and job-level contextual completion with and without abstraction. We also implement a statistical language model used as a baseline for comparison.

RQ_3: *To what extent "wrong" recommendations provided by GH-WCOM can be leveraged by developers?* RQ_3 gauges the extent to which "wrong" predictions (*i.e.,* recommendations different from the expected output) can still be useful to developers and thus worth being integrated into CI/CD pipelines after minor changes.

B.3.1 Data Collection and Analysis

To address RQ_1, we run the best-performing configuration for each pre-training strategy and scenario (NS_{task} and JC_{task}) against the test sets (Table B.1). Then, we compute the percentage of correct predictions, namely cases in which the models can synthesize completions identical to the expected target (*i.e.,* the code written by developers). We further assess the quality of the predictions generated using different pre-training strategies by relying on NLP (Natural Language Processing) metrics such as BLEU [PRWZ02] and ROUGE [Lin04].

BLEU score (Bilingual Evaluation Understudy) [PRWZ02] measures how similar the candidate (predicted) and reference (oracle) texts are. Given a size n, the candidate and reference texts are broken into n-grams and the algorithm determines how many n-grams of the candidate text appear in the reference text. The BLEU score ranges between 0 (the sequences are completely different) and 1 (the sequences are identical). We use the BLEU-4 variant as did in previous software engineering papers [WWJH21, WCP+22, TMM+22].

ROUGE (Recall-Oriented Understudy for Gisting Evaluation) is a set of metrics for evaluating both automatic summarization of texts and machine translation techniques [Lin04]. ROUGE metrics compare an automatically generated summary or translation with a set of reference summaries (typically, human-produced). We use the ROUGE-L which computes the length of the longest common subsequence between a generated and a reference sentence.

To answer RQ_2, we first select the best-performing models when supporting the completion of GitHub workflow with and without abstraction in both predictions scenario (NS_{task} and JC_{task}). Later, we assess the quality of the predictions using the same set of metrics (*i.e.,* correct predictions, BLEU, and ROUGE score) adopted in RQ_1. As there is no previous approach to compare GH-WCOM against, we implemented a baseline leveraging an n-gram model which is a specific actualization of a large class of techniques that assign probabilities to sequences of tokens (*i.e.,* Statistical-Language-Model [Gol17]). To train such a model we use the same set of instances used to fine-tune GH-WCOM without, however, any masked part. We experimented with three different values of n (*i.e.,* n=3, n=5, and n=7), with $n-1$ being the number of tokens on which the prediction of the next token is based upon. The best value for n ($n = 3$) has been found by running the models on the evaluation sets (results in our replication package [repc]).

The best model has then been run on the same test sets used for GH-WCOM's assessment. We do not compare GH-WCOM against the n-gram when job-level information is provided (JC_{task}), since, by construction, such a technique would not leverage the additional knowledge provided (*i.e.,* it only "looks" at the tokens preceding the ones to predict). To explain how predictions are generated with the 3-gram model, let us assume we are completing a piece of workflow having five tokens T, of which the last two are masked (M): $\langle T1, T2, T3, M4, M5\rangle$. We provide, as input to the model, T2 and T3 to predict M4, obtaining the model prediction P4. Then, we use T3 and P4 to predict M5 obtaining the predicted sentence $\langle T1, T2, T3, P4, P5\rangle$. While GH-WCOM autonomously decides when to stop predicting tokens, this is not the case for the n-gram model in our usage scenario. We thus defined two heuristics to stop generating tokens. First, we stop when the n-gram model does not generate any output token given the preceding n-1.

Second, we rely on the format in which we represent the instances in our datasets: Each instance is a JSON object and we trained all models to generate as output `{target}`, where the two delimiting curly brackets are the result of our JSON-like representation. Thus, we stop generating tokens when we reach a fully-balanced (*i.e.,* valid) JSON object for the test instance to predict (*i.e.,* the n-gram generated the "closing" curly bracket and the latter does not close a curly bracket opened in the predicted code but the JSON-related one).

We complement the quantitative evaluation by performing statistical tests aimed at as-

sessing whether GH-WCOM produces better recommendations as compared to the baseline. We use the McNemar's test [McN47] (with is a proportion test for dependent samples) and Odds Ratios (ORs) on the correct predictions both approaches (*i.e.,*GH-WCOM and n-gram) can generate when evaluated in the NS_{task} completion scenarios, working with both abstracted and raw tokens. We also statistically compare the distribution of the BLEU-4 (computed at statement level) and ROUGE, assuming a significance level of 95% and using the Wilcoxon signed-rank test [Wil45]. The (paired) Cliff's Delta (d) is used as effect size [GK05] and it is considered: negligible for $|d| < 0.10$, small for $0.10 \leq |d| < 0.33$, medium for $0.33 \leq |d| < 0.474$, and large for $|d| \geq 0.474$ [GK05]. Due to multiple comparisons for both statistical tests, we adjust p-values using Holm's correction procedure [Hol79].

As for RQ_3, we perform a twofold analysis. We first assess whether the confidence of the model in the generated predictions can be used as a reliable proxy of their "quality". T5 provides a *score* for each generated prediction which represents the log-likelihood of the prediction. For example, having a log-likelihood of -2 means that the prediction has a likelihood of 0.69 ($ln(x) = -2 \implies x = 0.69$). The likelihood can be interpreted as the confidence of the model about the correctness of the prediction on a scale from 0.00 to 1.00 (the higher the better). We split the predictions generated by T5 into ten buckets at steps of 0.1 (*i.e.,* the lowest confidence scenario groups the predictions having confidence between 0.0 and 0.1, the highest from 0.9 to 1.0) and report the percentage of correct and wrong predictions within each bucket. Then, given the positive results we achieved (as we will show, the confidence values are representative of the prediction quality), we randomly sample 384 cases of wrong predictions having a confidence $\geq$0.70, with 384 representing a statistically significant sample with a confidence level of 95% and confidence interval of $\pm$5%.

Each sample has been manually classified by two authors with one of the following labels:

1. A minor change is required to make the suggestion usable, *e.g.,* change an option or a value;
2. GH-WCOM has recommended the correct action/script command, yet with wrong arguments;
3. GH-WCOM has recommended the correct action/script command, yet with the wrong name;
4. The suggestion is completely wrong, *i.e.,*GH-WCOM recommendation is completely different from the ground truth.

In the labeling, the two involved authors achieved a Cohen's kappa [Coh60] of 0.72, indicating a *substantial agreement* when measuring inter-rater reliability for categorical items.

Conflicts, which occurred for 17.97% of inspected samples, have been solved through open discussion among the authors.

We report the percentage of predictions assigned to each label and discuss qualitative examples of wrong predictions which, however, might still be valuable for developers.

B.4 Study Results

RQ_1: How does GH-WCOM perform with different pre-training strategies? The results obtained by fine-tuning T5 using different pre-training strategies are presented in Table B.3. The table shows the model's performance in terms of correct predictions, BLEU-4, and ROUGE-LCS (F-measure). The best model for a given combination of task (*i.e.,* NS_{task} and JC_{task}) and evaluation metrics is reported in boldface. As expected, the $T5_{NO-PT}$ is outperformed by all pre-trained models, with 11.23% and 19.74% correct predictions for the NS_{task} and JC_{task} task, respectively, when working on raw code. When abstracting the dataset, the correct predictions for the $T5_{NO-PT}$ model improve—14.14% for NS_{task} and 26.96% for JC_{task} —while remaining the worst configuration.

Table B.3. Comparison among different pre-training strategies in terms of correct predictions, BLEU-4 and ROUGE-LCS (f-measure) computed at corpus level

Dataset	PT-Strategy	Correct predictions		BLEU 4		ROUGE-LCS	
		NS_{task}	JC_{task}	NS_{task}	JC_{task}	NS_{task}	JC_{task}
Raw	$T5_{NO-PT}$	11.23%	19.74%	13.70%	13.80%	44.0%	54.75%
	$T5_{YL}$	15.85%	24.51%	14.50%	24.10%	50.09%	61.20%
	$T5_{NL}$ [t5-]	**17.47%**	26.02%	**23.10%**	**29.60%**	**51.78%**	63.34%
	$T5_{NL+YL}$	17.33%	**26.35%**	16.40%	27.70%	51.74%	**63.58%**
Abstracted	$T5_{NO-PT}$	14.14%	26.98%	20.40%	24.20%	46.31%	59.92%
	$T5_{YL}$	19.81%	32.58%	13.80%	17.0%	53.30%	64.88%
	$T5_{NL}$ [t5-]	21.28%	33.84%	**28.40%**	**25.90%**	55.30%	66.51%
	$T5_{NL+YL}$	**21.36%**	**34.23%**	21.80%	18.40%	**55.37%**	**66.54%**

The results with pre-training (also) involving English documents ($T5_{NL}$ and $T5_{NL+YL}$) are always the best or the second-best in class, with performance very close to each other. Noteworthy, the usefulness of pre-training on English text when dealing with software-related tasks has been already documented in the literature [TDSS22] and is likely due to the vast presence of English terms in the code. Both $T5_{NL}$ and $T5_{NL+YL}$ models achieve the best performance on the abstracted workflows, with a percentage of correct predictions of around 21% for the NS_{task} task and 34% for the JC_{task} task.

Two observations can be made here. First, in the JC_{task} task, T5 is more successful thanks to the additional context provided before triggering the prediction (*i.e.,* the skeleton of the job defined by the developer—see Section B.2.3.2).

Second, the abstraction seems to substantially boost the model's performance, with ∼4% of additional correct predictions for the NS_{task} task and ∼8% in the JC_{task} task.

Table B.4 statistically compares the correct predictions achieved using the four different pre-training strategies for the two tasks and the two workflow representations (raw and abstract). Confirming what was said above, the performance of $T5_{NL}$ and $T5_{NL+YL}$ is always significantly better (adjusted p-value < 0.001) compared to the non-pre-trained models ($T5_{NO-PT}$) and to the ones pre-trained using `YAML` files only ($T5_{YL}$), with ORs going from 1.49 up to 4.88. The difference between $T5_{NL}$ and $T5_{NL+YL}$ is never statistically significant, showing that the two models are almost equivalent. This is an important finding because it means that an English pre-trained model can be simply fine-tuned to successfully accomplish the task (this is way less demanding than retraining the model).

Table B.4. Effect of different pre-training strategies on performance: results of McNemar's test.

Dataset	Task	Comparison	p-value	OR
Raw Tokens	NS_{task}	$T5_{NL}$ vs. $T5_{NO-PT}$	<0.001	4.88
		$T5_{NL}$ vs. $T5_{YL}$	<0.001	1.95
		$T5_{NL}$ vs. $T5_{NL+YL}$	0.50	1.05
		$T5_{NL+YL}$ vs $T5_{YL}$	<0.001	1.96
	JC_{task}	$T5_{NL}$ vs. $T5_{NO-PT}$	<0.001	3.60
		$T5_{NL}$ vs. $T5_{YL}$	<0.001	1.59
		$T5_{NL}$ vs. $T5_{NL+YL}$	0.10	0.88
		$T5_{NL+YL}$ vs $T5_{YL}$	<0.001	1.74
Abstracted Tokens	NS_{task}	$T5_{NL}$ vs. $T5_{NO-PT}$	<0.001	3.98
		$T5_{NL}$ vs. $T5_{YL}$	<0.001	1.75
		$T5_{NL}$ vs. $T5_{NL+YL}$	0.69	0.96
		$T5_{NL+YL}$ vs $T5_{YL}$	<0.001	1.88
	JC_{task}	$T5_{NL}$ vs. $T5_{NO-PT}$	<0.001	3.78
		$T5_{NL}$ vs. $T5_{YL}$	<0.001	1.49
		$T5_{NL}$ vs. $T5_{NL+YL}$	0.05	0.86
		$T5_{NL+YL}$ vs $T5_{YL}$	<0.001	1.70

The analysis of the BLEU and ROUGE metrics shown in Table B.3 confirms the above-described finding, *i.e.,* pre-training always helps, in particular when leveraging English sentences.

Answer to RQ_1. The pre-training boosts the performance of GH-WCOM. Pre-training with English text (possibly along with YAML files) helps to achieve the best performance.

In the following RQs we leverage the model pre-trained on English text and `YAML` files as the backbone of GH-WCOM.

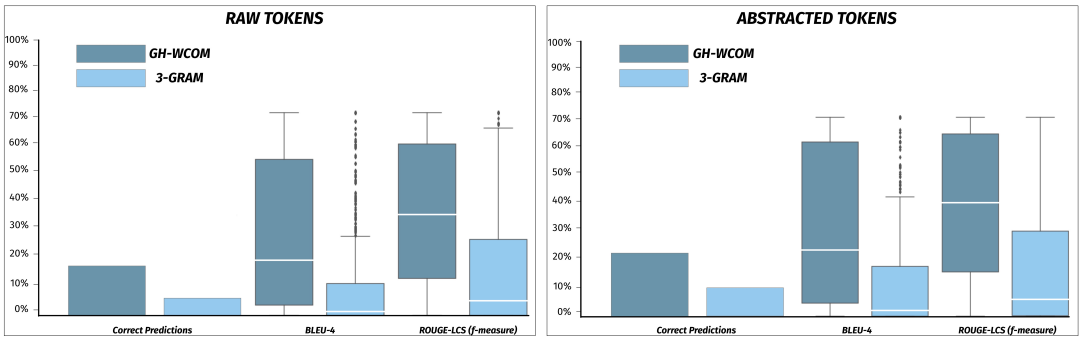


Figure B.4. Results achieved by GH-WCOM and the n-gram model when predicting actions for NS_{task}

RQ$_2$: How does GH-WCOM perform for different prediction scenarios? Fig. B.4 depicts the results achieved by GH-WCOM and the best-performing n-gram model (3-gram) in terms of correct predictions, BLEU-4 and ROUGE-LCS. Due to the technical limitations of the n-gram (*i.e.,* it only considers the $n-1$ preceding tokens when generating a prediction), such a comparison has been performed only for the NS_{task} task.

Table B.5 reports the results of the statistical comparison between the two in terms of adjusted p-value and OR (for correct predictions) and effect size (for BLEU and ROUGE). On both datasets, GH-WCOM achieves statistically significant better results than the baseline for all metrics. When looking at the correct predictions the gap is of ∼11% on the raw dataset (5.10% *vs* 17.33%) and ∼12% on the abstracted dataset (9.28% *vs* 21.36%). The OR is 17.69 (raw) and 13.76 (abstract). An OR of 13.76 indicates ∼13 times higher odds of obtaining a correct prediction using GH-WCOM. Even the comparisons in terms of BLEU and ROUGE show the superiority of GH-WCOM both visually (Fig. B.4) and statistically (Table B.5).

GH-WCOM achieves its best performance for the JC_{task} task, with 34.23% of correct predictions (see Table B.3), benefiting from the additional contextual information provided as input. Truly, one may question the usefulness of an approach that fails 66% of the times. Nevertheless, as a term for comparison, the DL-based approach recently proposed by Ciniselli *et al.* [CCP$^+$21] for block-level *Java* completion achieved ∼27% of correct predictions.

Table B.5. v s 3-gram model when generating recommendations for the NS_{task}

Dataset	Comparison	Metric	p-value	d	OR
Raw tokens	GH-WCOM vs. n-gram	Correct Predictions	<0.001	-	17.69
		BLEU-4	<0.001	0.51 (L)	-
		ROUGE-LCS	<0.001	0.52 (L)	-
Abstracted tokens	GH-WCOM vs. n-gram	Correct Predictions	<0.001	-	13.76
		BLEU-4	<0.001	0.49 (L)	-
		ROUGE-LCS	<0.001	0.50 (L)	-

Answer to RQ$_2$. GH-WCOM outperforms the n-gram baseline for the NS_{task} task on all the considered metrics. The gap in correct predictions is >11% on both the raw and the abstracted dataset. The best performances are achieved for the JC_{task} task (∼34% of correct predictions) thanks to the additional contextual information provided as input.

RQ$_3$: To what extent "wrong" recommendations provided by GH-WCOM can be leveraged by developers? Fig. B.5 depicts the relationship between the percentage of correct and wrong predictions when considering their confidence. Due to space limitations, we only focus our discussion on the most challenging scenario, namely NS_{task} , as the findings for JC_{task} are similar (complete results in [repd]). The orange line shows the percentage of correct predictions within each confidence interval, *e.g.,* 68.45% of predictions having confidence between 0.8 and 0.9 are correct when working with the raw code. In contrast, the red line shows the percentage of wrong predictions within each confidence bucket. Fig. B.5 shows a clear relationship between the confidence of the predictions and their likelihood of being correct. For example, out of the 1,076 predictions generated with confidence >0.9 in

the abstracted dataset, 959 (89.13%) are correct.

This result has an important practical implication: By setting a threshold on confidence, it would be possible to filter out recommendations likely to be false positives and only notify the developer when the model is quite confident about the generated prediction. As previously said, the results for the JC_{task} are in line with those discussed for NS_{task} . For example, 89.03% of the 2,908 predictions having confidence >0.9 are correct in the abstracted dataset. A similar percentage is achieved on the raw dataset (89.13%).

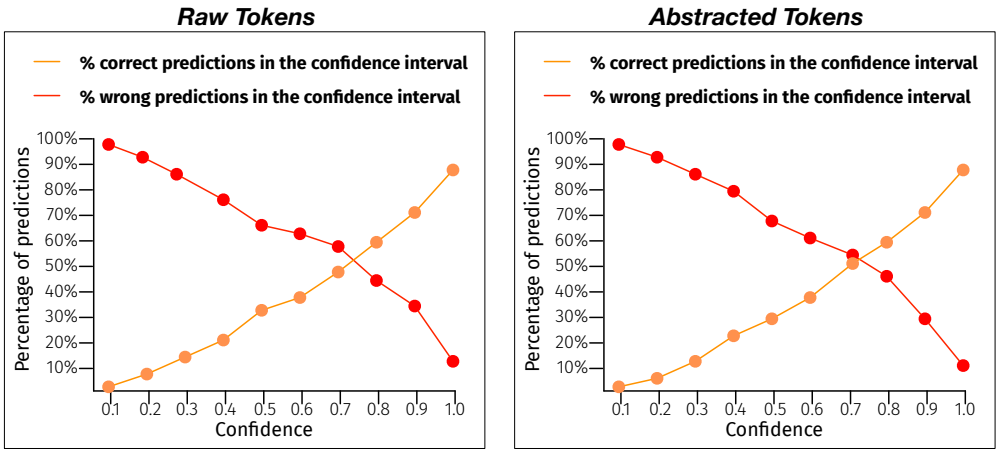


Figure B.5. Correct and wrong predictions by the confidence of GH-WCOM when generating recommendations for NS_{task}

Concerning the manual analysis of a sample of 384 completions "wrongly" predicted by GH-WCOM (*i.e.,* the prediction did not match the expected target), we found that: (i) 41.41% (159) are actually wrong, since the predicted code would implement a different behavior than the ground-truth; (ii) in 25.52% (98) of the cases, GH-WCOM suggested the correct action/script command yet with wrong arguments; (iii) 28.13% (108) of predictions would require minor changes, implying, on average, changing (*i.e.,* insertion and/or deletion) ∼11 characters in the recommended output in order to align with the ground truth; and (iv) 4.95% (19) feature a wrong or missing action name, *i.e.,* just missing documentation. While the complete results of our manual inspection are available in our replication package [repd], Fig. B.6 shows two concrete examples of the instances we inspected. The left part of Fig. B.6 ❶ shows an example in which the whole step is correctly predicted, with the exception of the name which is different from the expected one (`Set up Python` *vs* `Python`) but still meaningful. The right part ❷ depicts a case in which the only difference between the predicted and the expected step is the version of a specific action to use (`@v2` *vs* `@v3`). In both cases, the developer is still likely to benefit from the prediction.

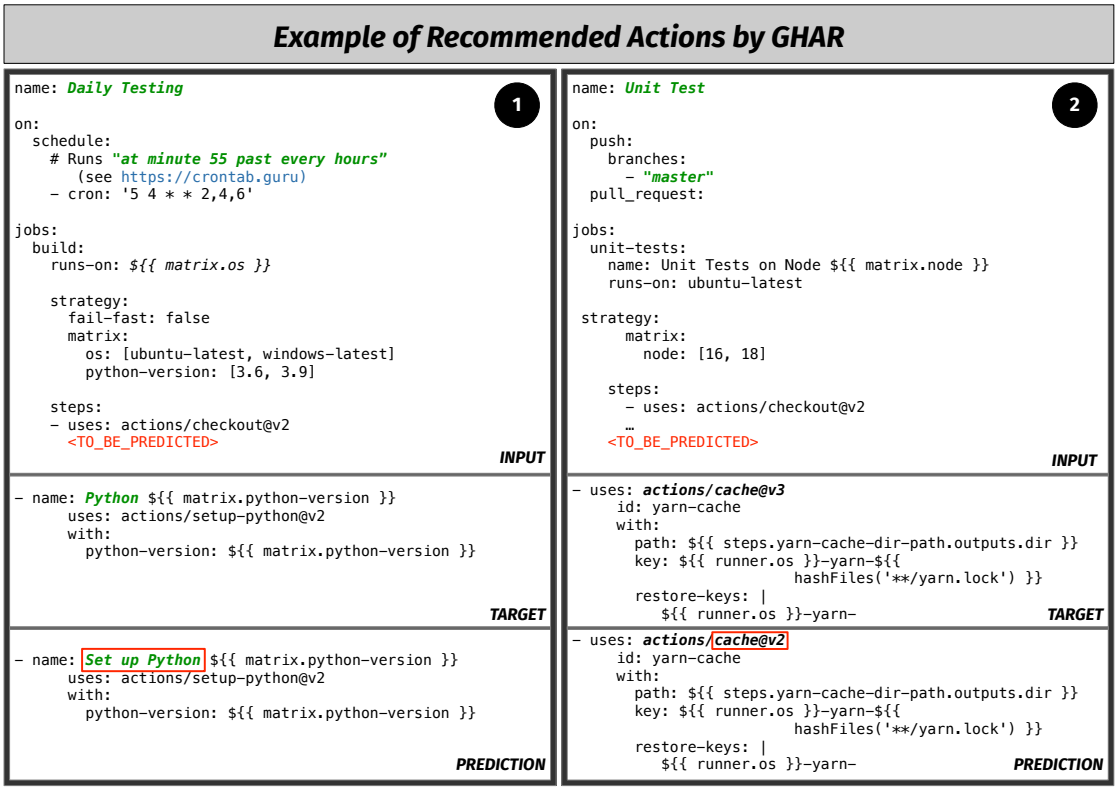


Figure B.6. Examples of GH-WCOM's recommended actions extracted from the manual investigation we performed

Answer to RQ$_3$. The confidence of the predictions can serve as a trustworthy indicator of their correctness when auto-completing GitHub workflows; ∼50% of predictions differing from the expected target but on which the model has high confidence could still be valuable for developers.

B.4.1 Why not just using a state-of-the-art chatbot or code recommender?

Large Language Models (LLMs) have opened up new possibilities even in the field of software engineering. One such application is GitHub Copilot [cop], developed by Microsoft using the OpenAI Codex model. Copilot is a state-of-the-art tool for recommending code completion and generation tasks. Similarly, OpenAI's ChatGPT [cha] showed remarkable performance in generating human-like text responses to prompts, even for code-related tasks.

We conducted a study to investigate the potential of these techniques for supporting auto-completion in GitHub workflows. We tested both tools on 60 instances in our test set by randomly selecting: (i) 15 workflows with the highest confidence score for which GH-WCOM provided correct predictions; (ii) 15 workflows with the highest confidence score for which GH-WCOM failed to provide meaningful recommendations; (iii) 15 workflows with the lowest confidence score for which GH-WCOM provided correct predictions; and (iv) 15 workflows with the lowest confidence score for which GH-WCOM failed to provide meaningful recommendations.

Concerning the high-confidence scenario, GitHub Copilot was able to provide correct recommendations for 7 of the 15 instances successfully predicted by GH-WCOM. For 2 instances, Copilot did not suggest any token, and for 6 instances, it provided incorrect recommendations. In contrast, when it came to the 15 instances for which GH-WCOM generated incorrect recommendations, Copilot correctly recommended only 2 of them and failed to provide meaningful recommendations for the remaining 13. Regarding ChatGPT, we observed that, out of the 15 instances correctly predicted by GH-WCOM, the chatbot can only suggest 4 meaningful GitHub workflow completions, while providing incorrect action elements/scripts for the remaining 11 instances.

We then tested ChatGPT on the instances where GH-WCOM failed, we found that for 13 out of 15 workflows, the recommended actions were incorrect, and, for 2 instances, ChatGPT was unable to respond to our query.

As for the GH-WCOM low-confidence instances, also Copilot and ChatGPT poorly performed on such instances. For the 15 successful predictions generated by GH-WCOM, Copilot succeeds in only 4 and ChatGPT in only 3 of them. Copilot and ChatGPT also fail in all 15 cases for which GH-WCOM provides a wrong output.

B.5 Threats to Validity

Construct validity. One potential threat arises from the collection of our dataset, as we excluded workflows longer than 1,024 tokens. As mentioned earlier, it is a common practice to limit the input size of DL models to manage training complexity effectively. We recognize that using different thresholds could yield varying results, and we acknowledge this as a potential limitation.

Another concern involves the extent to which the masking is representative of what programmers do during their tasks [HPGB19]. We have simulated two scenarios, NS_{task} and JC_{task}, representative of when developers write steps sequentially or code them after sketching their documentation. To evaluate the quality of the predictions, we used consolidated

measures such as the percentage of correct predictions, BLEU-4 [PRWZ02, RGL+20], and ROUGE score. Furthermore, we complemented such measures qualitative analyses.

In an attempt to help the model learning, we employed an abstraction schema in which five types of tokens are abstracted with special placeholders. The goal of our abstraction process was to identify a sort of upper-bound for the capabilities of our approach in a *best case* scenario, in which all tokes being *e.g.,* a path would be replaced with the same ⟨PATH⟩ placeholder. Such a simplification pushes more effort on the developer's side while, however, simplifying the learning, and thus representing an upper bound in terms of prediction performances (with the lower bound represented by the raw predictions). We acknowledge that alternative (and less extreme) solutions are possible; for example, distinct paths appearing within the same workflow could be abstracted with different placeholders (*e.g.,* ⟨PATH1⟩, ⟨PATH2⟩) with the model expected to use the same placeholder for related paths (*i.e.,* the same path appearing multiple times in the workflow). As part of our upcoming work agenda, we anticipate conducting user studies to assess different abstraction techniques as alternatives.

Internal validity. One key issue for DL models is the hyperparameter tuning, which we detailed in Section B.2.3.2. We are aware that we could not consider all possible (combinations of) values for that. Also, the performances of a T5 model could largely depend on how it has been pre-trained. To mitigate this threat, we have shown how GH-WCOM works by leveraging different pre-trainings.

Conclusion validity. To address the RQs, wherever appropriate we use suitable statistical tests (McNemar's test and Wilcoxon signed rank test) as well as effect size measures (OR and Cliff's delta). In the qualitative analysis of RQ_4, we computed and reported Cohen's kappa inter-rater agreement.

External validity. We experiment GH-WCOM with a $T5_{small}$ model. We acknowledge that our choice of the specific model architecture to use could affect the generalizability of our findings.

For example, larger T5 versions [RSR+20] could lead to different performance. We performed a minimal check of how scaling up the model could affect our findings. To this aim, we trained a $T5_{base}$ model [RSR+20] using the $T5_{NL+YL}$ setting and the same training process used for $T5_{small}$: We further pre-trained the publicly released $T5_{base}$ checkpoint (pre-trained on natural language) for 300k steps on YAML files and then fine-tuned it on the GitHub workflows. We used the same learning rate scheduler used for $T5_{base}$ (*i.e.,* ISQ-LR). The achieved results show that scaling up the model size from 60M to 220M parameters yields negligible improvements in comparison to $T5_{small}$. When employing a $T5_{base}$ architecture to recommend actions in the most demanding scenario (NS_{task}), the difference in correct predictions is a +0.18% (21.54%) and a +0.47% (17.80%) for the raw and abstracted datasets, respectively. When incorporating contextual information into the model (JC_{task}), similar conclusions arise (up to +0.67% of correct predictions). Furthermore, while we applied GH-WCOM for GitHub workflow completion, with proper training/fine-tuning, GH-WCOM could be applied to CI/CD pipelines developed with different technologies, *e.g.,* Jenkins or GitLab.

B.6 Conclusions and Future Work

This paper tackled the problem of automatically completing CI/CD pipeline scripts, and, in particular, GitHub workflows. We proposed , an approach based on T5 [RSR+20] pre-trained models to automatically recommend workflow completions in different scenarios, *i.e.,* predicting the next step (NS_{task}), or filling a workflow job given its textual documentation, *i.e.,* the `names` (JC_{task}).

Our empirical analysis found that (i) leveraging a pre-training involving English text (possibly complemented by `YAML` files) always helps, (ii) the performance of best models range from 17.47% (NS_{task} task) and 26.35% (JC_{task} task) for raw correct predictions, to 21.36% (NS_{task}) and 34.23% (JC_{task}) for abstracted correct predictions; and (iii) the model confidence correlates with the likelihood of generating a correct prediction. Finally, I s competitive for context-sensitive completion tasks when compared to LLM-based tools such as CoPilot [cop] and ChatGPT [cha].

Future work aims to experiment with alternative DL models, and, possibly, incorporate developers' feedback in the GH-WCOM's learning (*e.g.,* by using reinforcement learning).

C

Automated Variable Renaming: Are We There Yet?

Low-quality identifiers, such as meaningless method or variable names, are a recognized source of issues in software systems [LNB+19]. Indeed, choosing an expressive name for a program entity is not always trivial and requires both domain and contextual knowledge [Car82]. Even assuming a meaningful identifier is adopted in the first place while coding, software evolution may make the identifier not suitable anymore to represent a given entity. Moreover, the same entity used across different code components may be named differently, leading to inconsistent use of identifiers [LSM+17]. For these reasons, rename refactoring has become part of developers' routine [MHPB11], as well as a standard built-in feature in modern integrated development environments (IDEs). While IDEs aid developers with the mechanical aspect of rename refactoring, developers remain responsible for identifying low-quality identifiers and choosing a proper rename.

To support developers in improving the quality of identifiers, several techniques have been proposed [TR10, ABBS14, LSM+17, AZLY19]. Among those, data-driven approaches are on the rise [ABBS14, LSM+17, AZLY19]. This is also due to the recent successful application of these techniques in the code completion field [NNN+12, KZTC21, SDFS20, LLZJ20, CCP+21], which is a more general formulation of the variable renaming problem. Indeed, if a model is able to predict the next code tokens that a developer is likely to write (*i.e.,* code completion), then it can be used to predict a token representing an identifier. Nevertheless, strong empirical evidence about the performance ensured by such data-driven techniques for supporting developers in identifier renaming is still minimal.

In this work, we investigate the performance of three data-driven techniques in supporting automated variable renaming. We experiment with: (i) an *n*-gram cached language model [HD17]; (ii) the Text-to-Text Transfer Transformer (T5) model [RSR+20], and (iii) the Transformer-based model presented by Liu *et al.* [LLZJ20]. The *n*-gram cached language model [HD17] has been experimented against what were state-of-the-art deep learning models in 2017, showing its ability to achieve competitive performance in modeling source code. Thus, it represents a light-weight but competitive approach for the prediction of code tokens (including identifiers). Since then, novel deep learning models have been proposed such as,

for example, those based on the transformer architecture [VSP+17]. Among those, T5 has been widely studied to support code-related tasks [MSC+21, MAPB21, TMM+22, MPB22, WWJH21, CCP+21]. Thus, it is representative of transformer-based models exploited in the literature. Finally, the approach proposed by Liu *et al.* [LLZJ20] has been specifically tailored to improve code completion performance for identifiers, known to be among the most difficult tokens to predict. These three techniques provide a good representation of the current state-of-the-art of data-driven techniques for identifiers prediction.

All experimented models require a training set to learn how to suggest "meaningful" identifiers. To this aim, we built a large-scale dataset composed of 1,221,193 instances, where an instance represents a *Java* method with its related local variables. This dataset has been used to train, configure (*i.e.,* hyperparameters tuning), and perform a first assessment of the three techniques. In particular, as done in previous works related to the automation of code-related activities [AZLY19, TWB+19a, TPW+19, WTM+20, HLWM20, TPT+21], we considered a prediction generated by the models as correct if it resembles the choice made by the original developers (*i.e.,* if the recommended variable name is the same chosen by the developers). However, this validation assumes that the identifiers selected by the developers are meaningful, which is not always the case.

To mitigate this issue and strengthen our evaluation, we built a second dataset using a novel methodology we propose to increase the confidence in the dataset quality (in our case, in the quality of the identifiers in code). In particular, we mined variable identifiers that have been introduced or modified during a code review process (*e.g.,* as a result of a reviewer's comment). These identifiers result from a shared agreement among multiple developers, thus increasing the confidence in their meaningfulness. This second dataset is composed of 457 *Java* methods with their related local variables.

Finally, we created a third dataset aimed at simulating the usage of the experimented tools for rename variable refactoring: We collected 400 Java projects in which developers performed a rename variable refactoring. By doing so, we were able to mine 442 valid commits. For each commit c in our dataset, we checked-out the system's snapshot before (s_{c-1}) and after (s_c) the rename variable implemented in c. Given v the variable renamed in c, we run the three techniques on the code in s_{c-1} (*i.e.,* before the rename variable refactoring) to predict v's identifier.

Then, we check whether the predicted identifier is the one implemented by the developers in s_c. If this is the case, this means that the approach was able to successfully recommend a rename variable refactoring for v, selecting the same identifier chosen by developers.

Our quantitative analysis shows that the Transformer-based model proposed by Liu *et al.* [LLZJ20] is by far the best performing model in the literature for the task of predicting variable identifiers. This confirms the effort performed by the authors that aimed at specifically improve the performance of DL-based models in this task. This approach, named CugLM, can correctly predict the variable identifier in $\sim$ 63% of cases when tested on the large scale dataset we built. Concerning the other two datasets, the performance of all models drop, with CugLM still ensuring the best performance with $\sim$ 45% of correct predictions on both datasets.

We also investigate whether the "confidence of the predictions" generated by the three

models (*i.e.,* how "confident" the model are about the generated prediction) can be used as a proxy for prediction quality. We found that when the confidence is particularly high ($> 90\%$), the predictions generated by the models, and in particular by CugLM, have a very high chance of being correct (>80% on the large-scale dataset). This suggests that the recommendations generated such tools, under specific conditions (*i.e.,* high confidence) are ready to be integrated in rename refactoring tools.

We complement the study with a qualitative analysis aimed at inspecting "wrong" predictions to (i) see whether, despite being different from the original identifier chosen by the developer, they still represent meaningful identifiers for the variable provided as input; and (ii) distill lessons learned from these failure cases.

Concerning the first point, it is indeed important to clarify that even wrong predictions may be valuable for practitioners. This happens, for example, in the case in which the approach is able to recommend a valid alternative for an identifier (*e.g., surname* instead of *lastName*) or maybe even suggesting a better identifier, thus implicitly recommending a rename refactoring operation. Such an analysis helps in better assessing the actual performance of the experimented techniques.

Finally, we analyze the circumstances under which the experimented tools tend to generate correct and wrong predictions. For example, not surprisingly, we found that these approaches are effective in recommending identifiers that they have already seen used, in a different context, in the training set. Also, the longer the identifier to predict (*e.g.,* in terms of number of terms composing it), the lower the likelihood of a correct prediction.

Significance of research contribution. To the best of our knowledge, our work is the largest study at date experimenting with the capabilities of state-of-the-art data-driven techniques for variable renaming across several datasets, including two *high-quality datasets* we built with the goal of increasing the confidence in the obtained results. The three datasets we built and the code implementing the three techniques we experiment with are publicly available [repb]. Our findings unveil the potential of these tools as support for rename refactoring and help in identifying gaps that can be addressed through additional research in this field.

C.1 Data-driven Variable Renaming

In our study, we aim at assessing the effectiveness of data-driven techniques for automated variable renaming. We focus on three techniques representative of the state-of-the-art. The first is a statistical language model that showed its effectiveness in modeling source code [HD17]. The second, T5 [RSR+20], is a recently proposed DL-based technique already applied to address code-related tasks [MSC+21]. The third is the Transformer-based model presented by Liu *et al.* [LLZJ20] to boost code completion performance on identifiers.

Fig. C.1 depicts the scenario in which these techniques have been experimented. We work at method-level granularity: For each local variable v declared in a method m, we mask every v's reference in m asking the experimented techniques to recommend a suitable name for v. If the recommended name is different from the original one, a rename variable recommendation can be triggered.

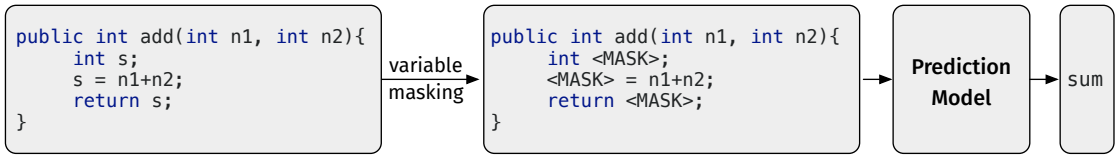


Figure C.1. Variable renaming scenario

We provide an overview of the experimented techniques, pointing the reader to the papers introducing them [HD17, RSR+20, LLZJ20] for additional details. Our implementations are based on the ones made available by the original authors of these techniques and are publicly available in our replication package [repb]. The training of the techniques is detailed in Section C.2.

C.1.1 N-gram Cached Model

Statistical language models can assess a probability of a given sequence of words. The basic idea behind these models is that the higher the probability, the higher the "*familiarity*" of the scored sequence. Such *familiarity* is learned by training the model on a large text corpus. An n-gram language model predicts a single word following the $n-1$ words preceding it. In other words, n-gram models assume that the probability of a word only depends on the previous $n-1$ words.

Hellendoorn and Devanbu [HD17] discuss the limitations of n-gram models that make them suboptimal for modeling code (*e.g.,* the unlimited vocabulary problem due to new words that developers can define in identifiers). To overcome these limitations, the authors present a *dynamic, hierarchically scoped, open vocabulary language model* [HD17], showing that it can outperform Recurrent Neural Networks (RNN) and LSTM in modeling code. While Karampatsis *et al.* [KS19] showed that DL models can outperform the cached n-gram model, the latter ensures good performance at a fraction of the DL models training cost, making it a competitive baseline for code-related tasks.

C.1.2 Text-To-Text-Transfer-Transformer (T5)

The T5 model has been introduced by Raffel *et al.* [RSR+20] to support multitask learning in Natural Language Processing. The idea is to reframe NLP tasks in a unified text-to-text format in which the input and output of all tasks to support are always text strings. For example, a single T5 model can be trained to translate across a set of different languages (*e.g.,* English, German) and identify the sentiment expressed in sentences written in any of those languages. This is possible since both these tasks (*i.e.,* translation and sentiment identification) are text-to-text tasks, in which a text is provided as input (*i.e.,* a sentence in a specific language for both tasks) and another text is generated as output (*i.e.,* the translated sentence or a label expressing the sentiment). T5 is trained in two phases: *pre-training*, which allows defining a shared knowledge-base useful for a large class of text-to-text tasks (*e.g.,* guessing masked words in English sentences to learn about the language), and *fine-tuning*, which specializes

the model on a specific downstream task (*e.g.,* learning the translation of sentences from English to German). As previously said, T5 already showed its effectiveness in code-related tasks [MSC+21]. However, its application to variable renaming is a premier. Among the T5 variants proposed by Raffel *et al.* [RSR+20] that mostly differ in terms of architectural complexity, we adopt the smallest one ($T5_{small}$). The choice of such architecture is driven by our limited computational resources. However, we acknowledge that bigger models have been shown to further increase performance [RSR+20].

C.1.3 Deep-Multi-Task code completion model

Liu *et al.* [LLZJ20] recently proposed the *Code Understanding and Generation pre-trained Language Model (CugLM)*, a BERT-based model for source code modeling. Albeit under-the-hood CugLM still features a Transformer-based network [VSP+17] as T5, such an approach has been specifically conceived to improve the performance of language models in identifiers, thus making it very suitable for our study on variable renaming. CugLM is pre-trained using three objectives. The first asks the model to predict masked identifiers in code (being thus similar to the one used in the T5 model, but focused on identifiers). The second task asks the model to predict whether two fragments of code can follow each other in a snippet. Finally, the third is a left-to-right language modeling task, in which the classic code completion scenario is simulated, *i.e.,* given some tokens (left part), guess the following token (right part).

Once pre-trained, the model is fine-tuned for code completion in a multi-task learning setting, in which the model has first to predict the type of the following token and, then, the predicted type is used to foster the prediction of the token itself. As reported by the authors, such an approach achieves state-of-the-art performance when it comes to predicting code identifiers.

C.2 Study Design

The *goal* is to experiment the effectiveness of data-driven techniques in supporting automated variable renaming. The *context* is represented by (i) the three techniques [HD17, RSR+20, LLZJ20] introduced in Section C.1 and (ii) three datasets we built for training and evaluating the approaches. Our study answers the following research question: ***To what extent can data-driven techniques support automated variable renaming?***

C.2.1 Datasets Creation

To train and evaluate the experimented models, we built three datasets: (i) the *large-scale dataset*, used to train the models, tune their parameters, and perform a first assessment of their performance; (ii) the *reviewed dataset* and (iii) the *developers dataset* used to further assess the performance of the experimented techniques. Our quantitative evaluation is based on the following idea: If, given a variable, a model is able to recommend the same identifier

name as chosen by the original developers, then the model has the potential to generate meaningful rename recommendations.

Clearly, there is a strong assumption here, namely that the identifier selected by the developers is meaningful. For this reason, we have three datasets. The first one (*large-scale dataset*) aims at collecting a high number of variable identifiers that are needed to train the data-driven models and test them on a large number of data points. The second one (*reviewed dataset*) focuses instead on creating a test set of high-quality identifiers for which our assumption can be more safely accepted: These are identifiers that have been modified or introduced during a code review process. Thus, more than one developer agreed on the appropriateness of the chosen identifier name for the related variable. Finally, the third dataset (*developers dataset*) focuses on identifiers that have been subject to a rename refactoring operation (*i.e.,* the developer put effort in improving the quality of the identifier through a refactoring). Again, this increases our confidence in the quality of the considered identifiers.

In this section, we describe the datasets we built, while Section C.2.2 details how they have been used to train, tune, and evaluate the three models.

C.2.1.1 Large-scale Dataset

We selected projects on GitHub [Gitb] by using the search tool by Dabic *et al.* [DAB21]. This tool indexes all GitHub repositories written in 13 different languages and having at least 10 stars, providing a handy querying interface [GHS] to identify projects meeting specific selection criteria. We extracted all *Java* projects having at least 500 commits and at least 10 contributors. We do so as an attempt to discard toy/personal projects. We decided to focus on a single programming language to simplify the toolchain building needed for our study. Also, we excluded forks to reduce the risk of duplicated repositories in our dataset.

Such a process resulted in 5,369 cloned Java projects from which we selected the 1,425 using Maven[1] [Mav] and having their latest snapshot being compilable. Maven allows to quickly verify the compilability of the projects, which is needed to extract information about types needed by one of the experimented models (*i.e., CugLM*). *CugLM* leverages identifiers' type information to improve its predictions. To be precise and comprehensive in type resolution, we decided to rely on the JavaParser library [Javnd], running it on compilable projects. This allows to resolve also types that are implemented in imported libraries. We provide the tool we built for such an operation as part of our replication package [repb].

We used *srcML* [CDM13] to extract from each *Java* file contained in the 1,425 projects all methods having $\#tokens \leq 512$, where $\#tokens$ represents the number of tokens composing a function (excluding comments). The filter on the maximum length of the method is needed to limit the computational expense of training DL-based models (similar choices have been made in previous works [TWB+19a, HLWM20, TPT+21], with values ranging between 50 and 100 tokens). All duplicate methods have been removed from the dataset to avoid overlap between training and test sets we built from them.

[1]Maven is a software project management tool that, as reported in its official webpage (`https://maven.apache.org`) "can manage a project's build, reporting and documentation from a central piece of information".

From these 1,425 repositories, we set apart 400 randomly selected projects for constructing the *developers dataset* (described in Section C.2.1.3). Concerning the remaining 1,025, we use ∼40% of them (418 randomly picked repositories) to build a dataset needed for the pre-training of the T5 [RSR+20] and of the *CugLM* model [LLZJ20] (*pre-training dataset*). Such a dataset is needed to support the pre-training phase that, as shown in the literature, helps deep learning models to achieve better performance when dealing with code-related tasks [TDS+20, TMM+22, MPB22, CCP+21]. Indeed, the pre-training phase conveys two major advantages summarized as follows: (i) once the model has been pre-trained, it can learn general representations and patterns of the language the model is working with, (ii) the pre-trained model yields to a more robust model initialization of the neural network weights that can then support the specialization phase (*i.e.,* fine-tuning).

The remaining 615 projects (*large-scale dataset*) have been further split into training (60%), evaluation (20%), and test (20%). The training set has been used to fine-tune the two DL-based models (*i.e.,* T5 and *CugLM*). This dataset, joined with the *pre-training dataset*, has also been used to train the *n*-gram model. In this way, all models have been trained using the same set of data, with the only difference being that the training is organized in two steps (*i.e.,* pre-training and fine-tuning) for T5 and *CugLM*, while it consists of a single step for the *n*-gram model.

For the T5 model, we used the evaluation set to tune its hyperparameters (Section C.2.2), since no previous work applied such a model for the task of variable renaming. Instead, for *CugLM* and *n*-gram we used the best configurations reported in the original works presenting them [HD17, LLZJ20]. Finally, the test set has been used to perform a first assessment of the models' performance.

Dataset	train	eval	test
pre-training dataset	500,414	-	-
large-scale dataset	394,574	176,944	149,261
reviewed dataset	-	-	457
developers dataset	-	-	442

Table C.1. Num. of methods in the datasets used in our study

Table C.1 shows the size of the datasets in terms of the number of extracted methods (*reviewed dataset* and *developers dataset* are described in the following).

C.2.1.2 Reviewed Dataset

Also in this case, we selected GitHub projects using the tool by Dabic *et al.* [DAB21]. Since the goal for the *reviewed dataset* is to mine code review data, we added on top of the selection criteria used for the *large-scale dataset* a minimum of 100 pull requests per selected project. Also in this case we only selected Maven projects having their latest snapshot successfully compiling. We then mined from the 948 projects we obtained information related to the code review performed in their pull requests. Let us assume that a set of files C_s is submitted for review. A set of reviewer comments R_c can be made on C_s possibly resulting in a revised

version of the code C_{r_1}.

Such a process is iterative and can consists of several rounds each one generating a new revised version C_{r_i}. Eventually, if the code contribution is accepted for merging, this concludes the review process with a set of C_f files. This whole process "transforms" $C_s \rightarrow C_f$. We use *srcML* to extract from both C_s and C_f the list of methods in them and, by performing a *diff*, we identify all variables that have been introduced or modified in each method as result of the review process (*i.e.,* all variables that were not present in C_s but that are present in C_f). We conjecture that the identifiers used to name these variables, being the output of a code review process, have a higher chance of representing high quality data that can be used to assess the performance of the experimented models.

Also in this case we removed duplicate methods both (i) within the *reviewed dataset*, and (ii) between it and the previous ones (*pre-training dataset* and training set of *large-scale dataset*), obtaining 457 methods usable as a further test set of the three techniques.

C.2.1.3 Developers' Dataset

We run *Refactoring miner* [TKD20] on the history of the 400 Java repositories we previously put aside. Refactoring miner is the state-of-the-art tool for refactoring detection in Java systems. We run it on every commit performed in the 400 projects, looking for commits in which a *Rename Variable refactoring* has been performed on the local variable of a method. This gives us, for a given commit c_i, the variable name at commit c_{i-1} (*i.e.,* before the refactoring) and the renamed variable in commit c_i. We use this set of commits as an additional test set (*developers dataset*) to verify if, by applying the experimented techniques on the c_{i-1} version, they are potentially able to recommend a rename (*i.e.,* they suggest, for the renamed variable, the identifier applied with the rename variable refactoring). After removing duplicated methods from this dataset as well (similarly, we also removed duplicates between *developers dataset*, *pre-training dataset*, and the training set in *large-scale dataset*), we ended up with 442 valid instances.

C.2.2 Training and Hyperparameters Tuning of the Techniques

C.2.2.1 N-gram model

The n-gram model has been trained on the instances in *pre-training dataset* $\cup$ *large-scale dataset*. This means that all Java methods contained in *pre-training dataset* and in the training set of *large-scale dataset* have been used for learning the probability of sequences of tokens. We use $n = 3$ since higher values of n have been proven to result in marginal performance gains [HD17].

C.2.2.2 T5

To pre-train the T5, we use a self-supervised task similar to the one by Raffel *et al.* [RSR$^+$20], in which we randomly mask 15% of code tokens in each instance (*i.e.,* a Java method) from the *pre-training dataset*, asking the model to guess the masked tokens. Such a training is

intended to give the model general knowledge about the language, such that it can perform better on a given *down-stream* task (in our case, guessing the identifier of a variable). The pre-training has been performed for 200k steps (corresponding to ∼13 epochs on our *pre-training dataset*) since we did not observe any improvement going further. We used a 2x2 TPU topology (8 cores) from Google Colab to train the model with a batch size of 128. As a learning rate, we use the *Inverse Square Root* with the canonical configuration [RSR$^+$20]. We also created a new *SentencePiece* model (*i.e.,* a tokenizer for neural text processing) by training it on the entire pre-training dataset so that the T5 model can properly handle the Java language. We set its size to 32k word pieces.

In order to find the best configuration of hyper-parameters, we rely on the same approach used by Mastropaolo *et al.* [MSC$^+$21]. Specifically, we do not tune the hyperparameters of the T5 model for the pre-training (*i.e.,* we use the default ones), because the pre-training itself is task-agnostic, and tuning may provide limited benefits. Instead, we experiment with four different learning rate schedulers for the fine-tuning phase. Since this is the first time T5 is used for recommending identifiers, we also perform an ablation study aimed at assessing the impact of pre-training on this task. Thus, we perform the hyperparameter tuning for both the pre-trained and the non pre-trained model, experimenting with the four configurations in Table C.2: constant (C-LR), slanted triangular (ST-LR), inverse square (ISQ-LR), and polynomial (PD-LR) learning rate. We experiment the same configurations for the pre-trained and the non-pretrained models, with the only difference being the $LR_{starting}$ and LR_{end} of the PD-LR. Indeed, in the non pre-trained model (ablation study), we had to lower those values to make the gradient stable (see Table C.2).

Table C.2. Hyperparameters tuning for the T5 Model

Learning Rate	**Parameters**	**Pre-Trained**	**No Pre-Trained**
C-LR	LR	0.001	0.001
ST-LR	$LR_{starting}$	0.001	0.001
	LR_{max}	0.01	0.01
	Ratio	32	32
	Cut	0.1	0.1
ISQ-LR	$LR_{starting}$	0.01	0.01
	Warmup	10,000	10,000
PD-LR	$LR_{starting}$	0.01	0.001
	LR_{end}	0.01	0.001
	Power	0.5	0.5

We fine-tune the T5 for 100k steps for each configuration. Then, we compute the percentage of correct predictions (*i.e.,* cases in which the model can correctly predict the masked variable identifier) achieved in the evaluation set. The achieved results reported in Table C.3 showed a superiority of the ST-LR (second column) for the non pre-trained model, while for the pre-trained model, the PD-LR works slightly better. Thus, we use these two scheduler in our study for fine-tuning the final models for 300k steps.

The fine-tuning of the T5 required some further processing to the *large-scale dataset*.

Table C.3. T5 hyperparameter tuning results

Experiment	C-LR	ST-LR	ISQ-LR	PD-LR
Pre-trained	30.74%	29.11%	30.77%	**30.80%**
No Pre-trained	21.18%	**27.56%**	26.08%	23.90%

Given a Java method m having n distinct local variables, we create n versions of it m_1, m_2, ..., m_n each one having all occurrences of a specific variable masked with a special token. Such a representation of the dataset allows to fine-tune the T5 model by providing it pairs (m_j, i_j), where m_j is a version of m having all occurrences of variable v_j replaced with a `<MASK>` token and i_j is the identifier selected by the developers for v_j. This allows the T5 to learn proper identifiers to name variables in specific code contexts. The same approach has been applied on the *large-scale dataset* evaluation and test set, as well as on the *reviewed dataset* and *developers dataset*. In these cases, an instance is a method with a specific variable masked, and the trained model is used to guess the masked identifier. Table C.4 reports the number of instances in the datasets used for the T5 model. Note that such a masking processing was not needed for the *n*-gram model nor for *CugLM*, since they just scan the code tokens during training, and they try to predict each code token sequentially during testing. Still, it is important to highlight that all techniques have been trained and tested on the same code.

	train	eval	test
large-scale dataset	1,122,864	521,779	437,384
reviewed dataset	-	-	457
developers dataset	-	-	442

Table C.4. Instances in the datasets used for training, evaluating, testing the T5 model

C.2.2.3 CugLM

To pre-train and fine-tune the *CugLM* model we first retrieved the identifiers' type information for all code in the *pre-training dataset* and *large-scale dataset*. Then, we leveraged the script provided by the original authors in the replication package [Cug] to obtain the final instances in the format expected by the model. For both pre-training and fine-tuning (described in Section C.1.3), we rely on the same hyper-parameters configuration used by the authors in the paper presenting this technique [LLZJ20].

C.2.3 Performance Assessment

We assess the performance of the trained models against the *large-scale test set*, the *reviewed dataset*, and the *developers dataset*. For each prediction made by each model, we collect a measure acting as "confidence of the prediction", *i.e.,* a real number between 0.0 and 1.0 indicating how confident the model is about the prediction. For the *n*-gram model, such

a measure is a transformation of the entropy of the predictions. Concerning the T5, we exploited the `score` function to assess the model's confidence on the provided input. The value returned by this function ranges from minus infinity to 0 and it is the log-likelihood (ln) of the prediction. Thus, if it is 0, it means that the likelihood of the prediction is 1 (*i.e.,* the maximum confidence, since $ln(1) = 0$), while when it goes towards minus infinity, the confidence tends to 0. Finally, CugLM outputs the *log-prob* for each predicted tokens. Hence, we normalize this value throught the *exp* function.

We investigate whether the confidence of the predictions represents a good proxy for their quality. If the confidence level is a reliable indicator of the predictions' quality (*e.g.,* 90% of the predictions having $c > 0.9$ are correct), it can be extremely useful in the building of recommender systems aimed at suggesting rename refactorings, since only recommendations with high confidence could be proposed to the developer. We split the predictions by each model into ten intervals, based on their confidence c going from 0.0 to 1.0 at steps of 0.1 (*i.e.,* first interval includes all predictions having $0 \leq c < 0.1$, last interval has $0.9 \leq c$). Then, we report for each interval the percentage of *correct predictions* generated by each model in each interval. To assess the performance of the techniques overall, we also report the percentage of correct predictions generated by the models on the entire test datasets (*i.e.,* by considering predictions at any confidence level).

A prediction is considered "correct" if the predicted identifier corresponds to the one chosen by developers in the *large-scale dataset* and in the *reviewed dataset*, and if it matches the renamed identifier in the *developers dataset*. However, a clarification is needed on the way we compute the correct predictions. We explain this process through Fig. C.2, showing the output of the experimented models, given an instance in the test sets.

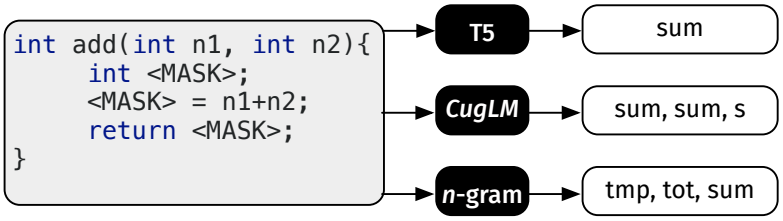


Figure C.2. Example of models' outputs for a test instance

The grey box in Fig. C.2 represents an example of an instance in the test set: a function having all references to a specific variable originally named `sum` masked. The T5 model, given such an instance as input, predicts a **single** identifier (*i.e.,* `sum`) for all three references of the variable. Thus, for the T5, it is easy to say whether the single generated prediction is equivalent to the identifier chosen by the developers or not. The *n*-gram and the *CugLM* model, instead, generate three predictions, one for each of the masked instances, despite they represent the same identifier.

Thus, for these two models, we use two approaches to compute the percentage of correct predictions in the test sets. The first scenario, named *complete-match*, considers the prediction as correct only if all three references to the variable are correctly predicted. Therefore, in the example in Fig. C.2, the prediction of the *CugLM* model (2 out of 3) is considered

wrong. Similarly, the *n*-gram prediction (1 out of 3 correct) is considered wrong. The second scenario, named *partial-match*, considers a prediction as correct if at least one of the instances is correctly predicted (thus, in Fig. C.2 both the *n*-gram and the *CugLM* predictions are considered correct).

We also statistically compare the performance of the models in terms of correct predictions: We use the McNemar's test [McN47], which is a proportion test suitable to pairwise compare dichotomous results of two different treatments. We statistically compare each pair of techniques in our study (*i.e.,* T5 *vs CugLM*, T5 *vs n*-gram, *CugLM vs n*-gram). To compute the test results for two techniques T_1 and T_2, we create a confusion matrix counting the number of cases in which (i) both T_1 and T_2 provide a correct prediction, (ii) only T_1 provides a correct prediction, (iii) only T_2 provides a correct prediction, and (iv) neither T_1 nor T_2 provide a correct prediction. We complement the McNemar's test with the Odds Ratio (OR) effect size. Also, since we performed multiple comparisons, we adjusted the obtained *p*-values using the Holm's correction [Hol79].

We also manually analyzed a sample of wrong predictions generated by the approaches with the goal of (i) assessing whether, despite being different from the original identifiers used by the developers, they were still meaningful; and (ii) identifying scenarios in which the experimented techniques fail. To perform such an analysis, we selected the top-100 wrong predictions for each approach (300 in total) in terms of confidence level. Three of the authors inspected all of them, trying to understand if the generated variable name could have been a valid alternative for the target one, while the fourth author solved conflicts. Note that, given 100 wrong predictions inspected for a given model, we do not check whether the other models correctly predict these cases. This is not relevant for our analysis, since the only goal is to understand the extent to which the "wrong" predictions generated by each model might still be valuable for developers.

Finally, we compare the correct and wrong predictions in terms of (i) the size of the context (*i.e.,* number of tokens composing the method and number of times a variable is used in the method), (ii) the length of the identifier in terms of number of characters and number of words composing it (as obtained through camelCase splitting); (iii) the number of times the same identifier appears in the training data. We show these distributions using boxplots comparing the two groups of predictions (*e.g.,* compare the length of identifiers in correct and wrong predictions). We also compare these distributions using the Mann-Whitney test [Con98] and the Cliff's Delta (d) to estimate the magnitude of the differences [GK05]. We follow well-established guidelines to interpret the effect size: negligible for $|d| < 0.10$, small for $0.10 \leq |d| < 0.33$, medium for $0.33 \leq |d| < 0.474$, and large for $|d| \geq 0.474$ [GK05].

C.3 Results Discussion

Table C.6 reports the results achieved by the three experimented models for each dataset in terms of correct predictions. For T5, both pre-trained and non pre-trained versions are presented. For the *n*-gram and CugLM, we report the results both when using *perfect match* and the *partial match* heuristic to compute the correct predictions, while this was not required for the T5 for which the results should be interpreted as *perfect matches*.

Before commenting on the results is also important to clarify that the cached *n*-gram model [HD17] exploited, as compared to the other two models, additional information due to the caching mechanism. Indeed, the caching allows the model to "look" at code surrounding the one for which tokens must be predicted (in our case, the method in which we want to predict the variable identifier). Given a method m_t in the test set, we provide its cloned repository as "test folder" to the *n*-gram model, in such a way that it is leveraged by the caching mechanism (we used the implementation from [HD17]).

Two observations can be easily made by looking at Table C.6. First, for T5, the pre-trained model works (as expected) better than its non pre-trained version. From now on, we focus on the pre-trained T5 in the discussion of the results. Second, consistently for all three datasets, CugLM outperforms the other models by a significant margin. In particular, when looking at the correct predictions (*complete match*), the improvement is +26% and +53% over T5 and *n*-gram, respectively, in the *large-scale dataset*. The gap is smaller but still substantial for the *reviewed dataset* (+15% and +44% for T5 and *n*-gram) and for the *developers dataset* (+22% and +43%). The difference in performance in favor of CugLM is always statistically significant (see Table C.5), with ORs going from 1.98 to 98.0. For example, on the *large-scale dataset* the ORs indicate that CugLM has 3.54 and 23.06 higher odds to generate a correct prediction as compared to the T5 and the *n*-gram model. These results confirm the suitability of the model proposed by Liu *et al.* [LLZJ20] when it comes to predicting code identifiers.

Table C.6 also shows that, as expected by construction, the percentage of correct predictions generated by CugLM and by the *n*-gram model increases when considering the *partial match* heuristic. However, for a fair comparison with the T5 model, we mostly focus our discussion on the *perfect match* scenario, that is also the one used in the computation of the statistical tests (Table C.5). The trend in performance is the same across the three datasets. However, the accuracy of all models drops on the *reviewed dataset* and on the *developers dataset*. Still, even in this scenario, CugLM is able to correctly recommend ∼50% of identifiers.

Table C.5. McNemar's test (adj. *p*-value and OR) considering complete matches as correct predictions. P-t=pre-trained

Dataset	**Test**	***p*-value**	**OR**
large-scale dataset	CugLM *vs* T5 (p-t)	< 0.001	3.54
	CugLM *vs* *n*-gram	< 0.001	23.06
	T5 (p-t) *vs* *n*-gram	< 0.001	6.08
reviewed dataset	CugLM *vs* T5 (p-t)	< 0.001	1.98
	CugLM *vs* *n*-gram	< 0.001	98.0
	T5 (p-t) *vs* *n*-gram	< 0.001	10.50
developers dataset	CugLM *vs* T5 (p-t)	< 0.001	3.90
	CugLM *vs* *n*-gram	< 0.001	32.50
	T5 (p-t) *vs* *n*-gram	< 0.001	24.25

Table C.6. Correct predictions: C-match indicates the *complete-match* heuristic, P-match the *partial-match*

large-scale dataset

#Instances	n-gram (c-match)		n-gram (p-match)		CugLM (c-match)		CugLM (p-match)		T5 (non pre-trained)		T5 (pre-trained)	
	#Correct	%Correct	#Correct	%Correct	#Correct	%Correct	#Correct	%Correct	#Correct	%Correct	#Correct	%Correct
437,384	46,126	10.54%	167,868	38.38%	277,595	63.46%	296,590	67.80%	153,708	35.14%	163,368	37.35%

reviewed dataset

#Instances	n-gram (c-match)		n-gram (p-match)		CugLM (c-match)		CugLM (p-match)		T5 (non pre-trained)		T5 (pre-trained)	
	#Correct	%Correct	#Correct	%Correct	#Correct	%Correct	#Correct	%Correct	#Correct	%Correct	#Correct	%Correct
457	20	4.37%	118	25.82%	214	48.35%	232	50.75%	142	31.07%	153	33.48%

developers dataset

#Instances	n-gram (c-match)		n-gram (p-match)		CugLM (c-match)		CugLM (p-match)		T5 (non pre-trained)		T5 (pre-trained)	
	#Correct	%Correct	#Correct	%Correct	#Correct	%Correct	#Correct	%Correct	#Correct	%Correct	#Correct	%Correct
442	8	1.90%	66	14.93%	197	45.02%	209	47.28%	87	19.68%	101	22.85%

Fig. C.3 depicts the relationship between the percentage of correct predictions and the confidence of the models. Similarly, to Fig. C.4, the orange line represents the *n*-gram model, while the purple and red lines represent CugLM and the T5 pre-trained model, respectively. Within each confidence interval (*e.g.,* 0.9-1.0) the line shows the percentage of correct predictions generated by the model (*e.g.,* ∼80% of predictions having a confidence higher than 0.9 are correct for CugLM in the *large-scale dataset*). The achieved results show a clear trend for all models: Higher confidence corresponds to higher prediction quality. The best performing model (CugLM) is able, in the highest confidence scenario, to obtain 66% of correct predictions on the *developers dataset*, 71% on the *reviewed dataset*, and 82% on the *large-scale dataset*. These results have a strong implication for the building of rename refactoring recommenders on top of these approaches: Giving the possibility to the user (*i.e.,* the developer) to only receive recommendations when the model is highly confident can discard most of the false positive recommendations.

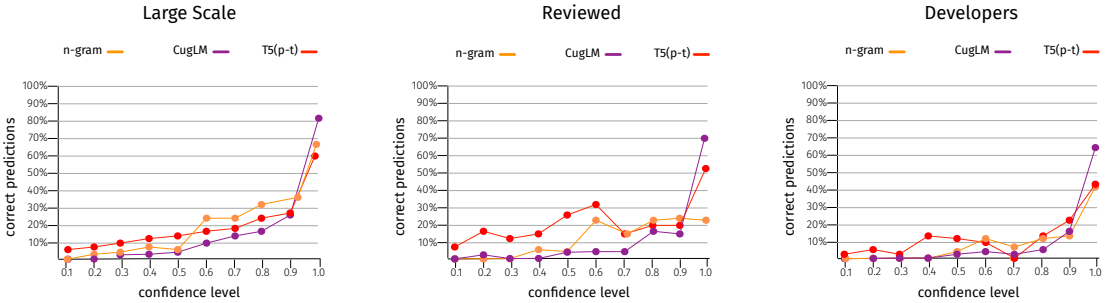


Figure C.3. Percentage of correct predictions by confidence level

Concerning the manual analysis we performed on 100 wrong recommendations generated by each model on the *large-scale dataset*, a few findings can be distilled. First, the three authors observed that the T5 was the one more frequently generating, in the set of wrong predictions we analyzed, identifiers that were meaningful in the context in which they were proposed (despite being different from the original identifier used by the developers). For example, `value` was recommended instead of `number` or `harvestTasks` instead of `tasks`. The three authors agreed on 31 meaningful identifiers proposed by the T5 in the set of 100 wrong predictions they inspected. Surprisingly, this was not the case for the other two models, despite the great performance we observed for CugLM. However, a second observation we made partially explains such a finding: We found that several failure cases of CugLM and of the *n*-gram model are due the recommendation of identifiers already used somewhere else in the method and, thus, representing wrong recommendations. We believe this is due to the different prediction mechanism adopted by the T5 as compared to the other two models. As previously explained, the T5 generates a single prediction for all instances of the identifier to predict, thus considering the whole method as a context for the prediction and inferring that identifiers already used in the context should not be recommended.

The other two models, instead, scan the method token by token predicting each identifier instance in isolation. This means that if an identifier x is used for the first time in the method after the first instance of the identifier p to predict (*e.g., p* appears in line 2 while x appears

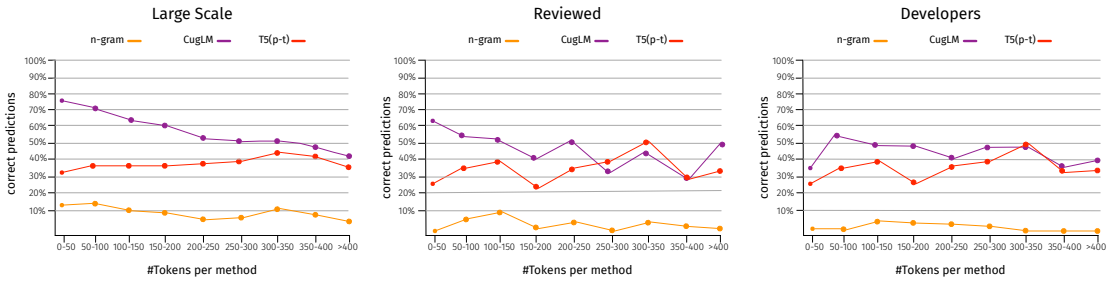


Figure C.4. Percentage of correct predictions by tokens per method

in line 7), the existence of x is not considered when generating the prediction for p. This reduces the information available to CugLM and to the n-gram model. Also, CugLM has limitations inherited from the fixed size of its vocabulary set to the 50k most frequent tokens [LLZJ20], which are the only ones the model can predict. This means that CugLM is likely to fail when dealing with rare identifiers composed by several words. The T5, using the SentencePiece tokenizer, can instead compose complex identifiers.

Finally, Fig. C.5 shows the comparison of five characteristics between correct (in green) and wrong (in red) predictions. The comparison has been performed on the three datasets (see labels on the right side of Fig. C.5), and for correct/wrong predictions generated by the three models (see labels on the left side of Fig. C.5). The characteristics we inspected are summarized at the top of Fig. C.5. For each comparison (*i.e.,* each pair of boxplots in Fig. C.5) we include a $*$ if the Mann-Whitney test reported a significant difference (p-value < 0.05) and, if this is the case, the magnitude of the Cliff's Delta is reported as well.

Concerning the length of the target identifier (*#Characters per identifier* and *#Tokens per identifier*), the models tend to perform better on shorter identifiers, with this difference being particularly strong for CugLM. Indeed, this is the only model for which we observed a large effect size in the length of identifiers between correct and wrong predictions. Focusing for example on the length in terms of number of tokens (*#Tokens per identifiers*), it is clear that excluding rare exceptions, CugLM mostly succeeds for one-word identifiers. This is again likely to be a limitation dictated by the fixed size of the vocabulary (50k tokens) that cannot contain all possible combinations of words used in identifiers.

The size of the coding context (*i.e.,* method) containing the identifier to predict (*#Tokens per method*) does not seem to influence the correctness of the prediction, with few significant differences accompanied by a negligible or small effect size. This is also visible by looking at Fig. C.4, which portrays the relationship between the percentage of correct predictions and the number of tokens composing the input method. The orange line represents the n-gram model, while the purple and red lines represent CugLM and the T5 pre-trained model, respectively. Within each interval (*e.g.,* 0-50), the line shows the percentage of correct predictions generated by the model for methods having a tokens length falling in that bucket. The only visible trend is that of *CugLM* on the *large-scale dataset*, which shows a clear downward trend in correct predictions with the increase in length of the input method. This is indeed the only scenario for which the statistical tests reported a significant differences in

Figure C.5. Characteristics of correct (green) and wrong (red) predictions

the method length of correct and wrong predictions with a small effect size (in all other cases, the difference is not significant or accompanied by a small effect size).

Differently, identifiers appearing in the training set tend to help the prediction (*#Overlapping identifiers within the training set*). This is particularly true for CugLM (large effect size on all datasets), since its vocabulary is built from the training set. The boxplot for the wrong predictions is basically composed only by outliers, with its third quartile equal 0. This indicates that the predictions on which CugLM fails are usually those for identifiers never seen in the training set.

Finally, the number of times that an identifier to predict appears in the context (*i.e.,*

#Occurences of identifier within methods), only has an influence for the T5 on the *large-scale dataset*. However, there is no strong trend to discuss for this characteristic.

C.3.1 Implications of our Findings

Our findings have implications for practitioners and researchers. For the first, our results show that modern DL-based techniques presented in the literature may be already suitable to be embedded in rename refactoring engines. Clearly, they still suffer of limitations that we will discuss later. However, especially when the confidence of their predictions is high, the generated identifiers are often meaningful, matching the ones chosen by developers.

In terms of research, there are a number of improvements these tools can benefit from. First, we noticed that the main weakness of the strongest approach we tested (*i.e.,* CugLM) is the fixed vocabulary size. Such a problem has been addressed in other models using tokenizers such as byte pair encoding [Gag94] or the SentencePiece tokenizer exploited by the T5. Integrating these tokenizers in CugLM (or similar techniques) could help in further improving performance. Second, we noticed that several false-positive recommendations could be avoided by just integrating into the models more contextual information.

For example, if the model is employed to recommend an identifier in a given location l, other identifiers having l in their scope do not represent a viable option, since they are already in use. Similarly, the integration of type information in CugLM demonstrated the boost of performance that can be obtained when the prediction model is provided with richer data. Also, the employed models are predicting identifier names without exploiting information such as (i) the original identifier name that could be improved via a rename refactoring, and (ii) the naming convention adopted in the project. Augmenting the context provided to the models with such information might substantially boost their prediction performance.

Finally, while we performed an extensive study about the capabilities of data-driven techniques for variable renaming, our experiments have been performed in an "artificial" setting. The (mostly positive) achieved results encourage the natural next step represented by case studies with developers to assess their perceived usefulness of these techniques.

C.4 Threats to Validity

Construct validity. Our study is largely based on one assumption: The identifier name chosen by developers is the correct one the models should predict. We addressed this threat when building two of our datasets: (i) we ensure that the variable identifiers in the *reviewed dataset* have been checked in the context of a code review process involving multiple developers; (ii) we built *developers dataset* by looking for identifiers explicitly renamed by developers. Thus, it is more likely that those identifiers are actually meaningful.

Internal validity. An important factor that influences DL performance is hyperparameters tuning. Concerning T5, for the pre-training phase we used the default T5 parameters selected in the original paper [RSR+20] since we expect little margin of improvement for such a task-agnostic phase. For the fine-tuning, due to feasibility reasons, we did not change the model architecture (*e.g.,* number of layers), but we experimented with different learning

rates-scheduler as did before by Mastropaolo *et al.* [MSC+21]. For the other two techniques we relied on the parameters proposed in the papers presenting them.

External validity. While the datasets used in our study represent hundreds of software projects, the main threat in terms of generalizability is represented by the focus on the *Java* language. It is important to notice that the experimented models are language agnostic, but would require the implementation of different tokenizers to support specific languages.

C.5 Conclusions and Future Work

We presented a large-scale empirical study aimed at assessing the performance of data-driven techniques for variable renaming. We experimented with three different techniques, namely the *n*-gram cached language model [HD17], the T5 model [RSR+20], and the Transformer-based model presented by Liu *et al.* [LLZJ20]. We show that DL-based models, especially when considering predictions they generate with high confidence, represent a valuable support for variable rename refactoring. Our future research agenda is dictated by the implications discussed in Section C.3.1.

D

Unveiling ChatGPT's Usage in Open Source Projects: A Mining-based Study

Recommender systems for software engineers have been defined by Robillard *et al.* [RWZ10] as:

> *"Software applications that provide information items estimated to be valuable for a software engineering task in a given context"*

Over the years, researchers have developed various forms of recommenders, aimed at suggesting relevant code elements for a given task [NNN+12, ABBS15, WAN+21], helping to fix bugs [XLD+17, TWB+19a, BSPC19, MH21b] and vulnerabilities [HLX+17, GWD+21, CKM22], and even to automatically document software systems [SPVS11, MBP+15, ABLL21].

In the last decade, the increasing gain of maturity and improvement of deep learning architectures, the availability of hardware infrastructures, and of data from forges such as GitHub has opened the road towards the development of recommenders able not only to better solve the aforementioned problems, but also to perform tasks for which no recommender was previously thought in the past, including generating entire code blocks [SDFS20, CCP+21], automatically reviewing source code [TPT+21, TWB+19a, LLG+22a, TPT22], or generating scenarios for automatically reproducing issues [ZSL+22, FMB+23, BCH+23].

The advent of Large Language Models (LLMs) and, lately, of LLM-based chat bots such as ChatGPT [cha] has opened new development landscapes. In such a context, a developer can, from a single tool, receive help for a wide number of tasks: one can ask ChatGPT to design an architecture, write or complete source code to achieve a given task, review and possibly refactor/optimize existing code, repair bugs, generate tests, and so on. In other words, today's ChatGPT and tomorrow's similar tools from other providers will gradually become the main source of help for developers, essentially replacing what a colleague, user manuals, the Web (including forums such as Stack Overflows) and, recently, code completion specialized tools such as GitHub Copilot [cop], have done so far.

Given such a scenario, it would be worthwhile understanding how developers have leveraged ChatGPT so far to achieve different goals. Certainly, this goal could possibly be achieved

through interviews and survey questionnaires. However, we have decided to follow a radically different approach, *i.e.,* by mining traces of ChatGPT usages in GitHub: commit messages, issues, and pull requests (PRs).

This is because on the one hand, we could observe how developers "admit" the use of ChatGPT in their (open-source) projects, but, also, how such code is being reviewed before being merged.

To conduct our study, we first mined all commits, issues, and PRs from GitHub that match the keyword "ChatGPT ". Then, we extracted n-grams surrounding the word ChatGPT and manually reviewed them for further filtering. This allowed us to filter the initial sample to reduce the chance of false positives, *e.g.,* an issue mentioning ChatGPT but not using it for the automation of a task. Then, we performed an open coding based on card sorting [Spe09] on all 1,501 candidate instances (*i.e.,* commit/issue/PR) we identified, classifying the ChatGPT purpose for each instance (*i.e.,* why it was used) or discarding the instance as a false positive.

As a result, we obtained a taxonomy of purposes for using ChatGPT in the automation of a software-related task. The taxonomy features seven root categories and 52 categories in total.

For each category, we discuss typical use cases, as well as implications for practitioners and researchers. Also, we highlight and discuss scenarios for which the use of ChatGPT turned out to be failing, counterproductive, or risky for the given activity.

All data used in our study is publicly available [TMP+23].

D.1 Study Design

The *goal* of the study is to unveil the purposes for which LLM recommenders are used to support the development of open-source projects. The *context* consists of ChatGPT, as a representative of state-of-the-art LLMs, and of 1,501 manually inspected commits, PRs, and issues sampled from open source projects hosted on GitHub. Our study aims at answering the following research question:

> *What are the software-related tasks for which developers document the support received by ChatGPT?*

We answer this research question by mining, from development artifacts, traces of ChatGPT usages. We focus on artifacts for which it is possible to perform keyword-matching queries on GitHub. As such, we search for commit messages, PRs, and issues mentioning ChatGPT in their textual content (Section D.1.1). We do not consider GitHub discussions, as we are interested to analyze text directly traceable to software artifacts. Then, we manually inspect 1,501 instances with the goal of categorizing the task(s) supported by ChatGPT (if any) in each of them (*e.g., generate tests*, *code review*) — see Section D.1.2. The obtained categories of tasks have then been used to derive a taxonomy of tasks supported by ChatGPT. Such a taxonomy provides (i) developers with a comprehensive catalog of usage scenarios in which LLM recommenders can be leveraged; and (ii) researchers with software-related tasks

which could benefit from automation, possibly through specialized solutions to be developed rather than via generic LLM recommenders such as ChatGPT.

In the following, we detail the steps behind our study design.

D.1.1 Mining Candidate Instances

The goal of this step is to identify commits, PRs, and issues in which ChatGPT has been *likely* used to support one or more tasks. False positive instances (*e.g.,* instances in which ChatGPT was mentioned but not actually leveraged) will be discarded in a later stage.

We started by querying—on June 12, 2023—the GitHub APIs to identify all commits, PRs, and issues containing the word "ChatGPT". For commits, the search was performed on the commit message/body, while for PRs and issues the target was their title and description. The output of this step were 186,425 commits, 15,629 PRs, and 31,934 issues (233,988 overall instances). By inspecting the retrieved instances, we noticed a predominance of false positives, mostly due to projects which integrate ChatGPT (*i.e.,* use the ChatGPT APIs) to offer features to their users (*e.g.,* a chat bot) rather than using it for automating software-related tasks.

We then performed a first filtering to automatically discard as many false positives as possible. To this aim, we extracted from the collected instances all forward/backward 2-grams and 3-grams containing the word "ChatGPT". For example, let us assume that a commit message features the sentence: "*Implemented matrix transposition with the help of ChatGPT*". In this case, we extract the following backward *n*-grams: "*of ChatGPT*", and "*help of ChatGPT*". Instead, a PRs titled *Used ChatGPT to implement tests* will result in the backward 2-gram "*used ChatGPT*" and in the forward *n*-grams "*ChatGPT to*" and "*ChatGPT to implement*".

We then sorted all extracted *n*-grams in ascending order of frequency, and inspected those appearing in at least 0.02% of the instances (>1k instances). We classified each *n*-gram as likely indicating the ChatGPT support in a task (*e.g.,* "*ChatGPT to generate*") or as likely indicating false positives (*e.g.,* "*ChatGPT API integration*"). This resulted in a set of 34 relevant *n*-grams available in our replication package [TMP+23]. We excluded all instances not containing at least one of such *n*-grams, and all those belonging to GitHub repositories having less than 10 stars in an attempt to filter out toy projects. Finally, we removed duplicates (*e.g.,* duplicated commits due to forked repositories) obtaining a final set of 1,501 instances, distributed as follows: 527 commits, 327 PRs, and 647 issues. The 1,501 manually analyzed issues, commits, and PRs belong to 732 different projects.

D.1.2 Manual Analysis and Taxonomy Definition

The goal of the manual analysis was to characterize within each of the 1,501 instances the task(s) (partially) automated using ChatGPT. Five authors (from now on evaluators) were involved in the manual inspection. Each instance has been independently inspected by two evaluators. The whole process was supported by a web app we developed that implemented the required logic and provided a handy interface to categorize the instance. For each instance, the evaluator was presented with: (i) the metadata as returned by the GitHub APIs (*e.g.,* for a commit: its author, message, body, date, etc.); (ii) the *n*-gram that was matched

in that specific instance (*e.g.,* "*ChatGPT to generate*"); and (iii) the link to the instance on GitHub for an easier inspection.

The categorization required the assignment of one or more lebels to an instance, describing the automated task(s) (*e.g., refactoring code*, *write documentation*). In case the manual inspection revealed that ChatGPT was not actually used to automate software-related tasks, the instance was discarded.

Since there are no documented taxonomies of software-related tasks automated with the support of LLM recommenders, we followed an open coding strategy [Spe09].

Specifically, each evaluator could introduce a new label, as they felt it was needed to properly describe the automated task(s). After the label was added, it became available, through the web app, to the other evaluators. While this goes against the notion of open coding, in a scenario in which there are no pre-defined categories this helps to reduce the chance of multiple evaluators defining similar labels to describe the same task while not introducing a substantial bias in the process.

It is important to mention that the labeling process has not been performed in a single shot, but rather in three rounds each involving roughly $\frac{1}{3}$ of the instances to inspect. At the end of each round the authors met to revise the set of labels defined up to that moment by (i) renaming unclear labels; (ii) merging similar labels, *i.e.,* labels describing the same automated task but with different wordings; and (iii) agreeing on irrelevant labels, actually indicating instances to discard (*i.e.,* unrelated to tasks supported by ChatGPT).

Once all 1,501 instances have been inspected by two evaluators, we solved conflicts. As for the relevance labeling, we found conflicts in 17% of the cases, with Cohen's $k = 0.64$, which is considered a strong agreement [Coh60].

For what concerns the (open-coded) categories, we found differences in 380 cases (∼25% of instances). While such a percentage may look high, this can be easily explained by two design choices. The first is the already mentioned lack of pre-defined categories. This implies that two evaluators defining semantically equivalent but different labels to describe an automated task (*e.g., create tests vs test writing*) would generate a conflict. The second concerns our conservative definition of conflicts: We considered an instance as a conflict if two evaluators assigned a different set of labels to the instance, *even if the two sets partially overlapped*. Conflicts also arose if one of the two evaluators discarded the instance as a false positive while the other labeled it. Each conflict has been inspected in pairs by two additional evaluators, who discussed and solved it. In the end, 467 instances were kept and classified, distributed as follows: 165 commits, 159 PRs, and 143 issues. The 467 classified instances belong to 358 projects, having [min=8, 1Q=60, median=444, 3Q=3,075, max=179,567] stars, and [min=0, 1Q=15, median=77.5, 3Q=535, max=89,252] forks.

The 45 labels defined through the above-described process have been used to build a hierarchical taxonomy of software-related tasks for which ChatGPT provided (partial) automation. Two of the authors created a preliminary version of the taxonomy which has then been refined in two rounds by collecting the feedback of all five authors involved in the labeling.

D.2 Results Discussion

Fig. D.1 depicts the taxonomy of tasks automated via ChatGPT. The taxonomy is composed of seven trees, each grouping together related tasks: *feature implementation/enhancement*, *process*, *learning*, *generating/manipulating data*, *development environment*, *software quality*, and *documentation*. The numbers attached to each task T_i indicate, from the right to the left, the number of commits, issues, and PRs in which we found evidence of T_i's automation using ChatGPT. For example, we found a total of 110 instances (43 commits, 29 issues, and 38 PRs) in which ChatGPT has been used to automate the implementation or enhancement of a feature.

Note that the sum of the number of instances in all tasks is greater than the total number of valid instances we inspected (467), since one instance may have required the support of ChatGPT for multiple tasks, *e.g., generating tests* and their related *code comments*. Also, note that the number of instances in a parent category is not always the sum of the instances in its child categories. For example, consider the *software quality* → *fixing* → *supporting debugging* category: Such a task has been automated in 12 instances (11 issues and 1 PRs) and has one child category named *writing code to reproduce a bug*, automated in 3 issues. The reason for such a discrepancy is that in 12 instances it was clear that ChatGPT has been used to support the debugging process, but only in three of those cases the classification could be even more precise and refer to the specific task of helping to reproduce a bug.

In the following, we discuss the seven main categories of automated tasks by reporting qualitative examples and discussing implications for practitioners (see 💻 icon) and researchers (💡). We also showcase ChatGPT's limitations when used for the automation of the related tasks (⚠). Due to the lack of space, we do not discuss all 52 categories in our taxonomy, but only the main ones. However, our replication package [TMP+23] provides the complete dataset reporting, for each category, the instances assigned to it.

D.2.1 Feature implementation/enhancement

This category features tasks related to the usage of ChatGPT as a support for implementing and enhancing software features. We start by commenting two of its related, but differing, subcategories: *implementing a new feature* and *prototyping*. The former refers to the usage of ChatGPT as a support to implement a specific *part* of a feature that the developer is working on. This means that the developer delegates the implementation of a specific functionality, which is then manually integrated with the rest of the code needed for the feature. An example is the PR #37233 from the `woocommerce` project [pr:23a], in which the PR author states: "*I wrote a Python script with ChatGPT to parse csv files since we need to update this payment list quarterly*".

The latter, instead, refers to the usage of ChatGPT as a way to quickly implement either (i) a complete feature that can be used as a starting prototype for reasoning about the addition of the new feature in the project, or (ii) the whole starting version of a project, on which developers can work and build on top. An example of the first scenario is the PR #73 from the `nix-gaming` project [pr:23b] in which the contributor proposes the addition of an

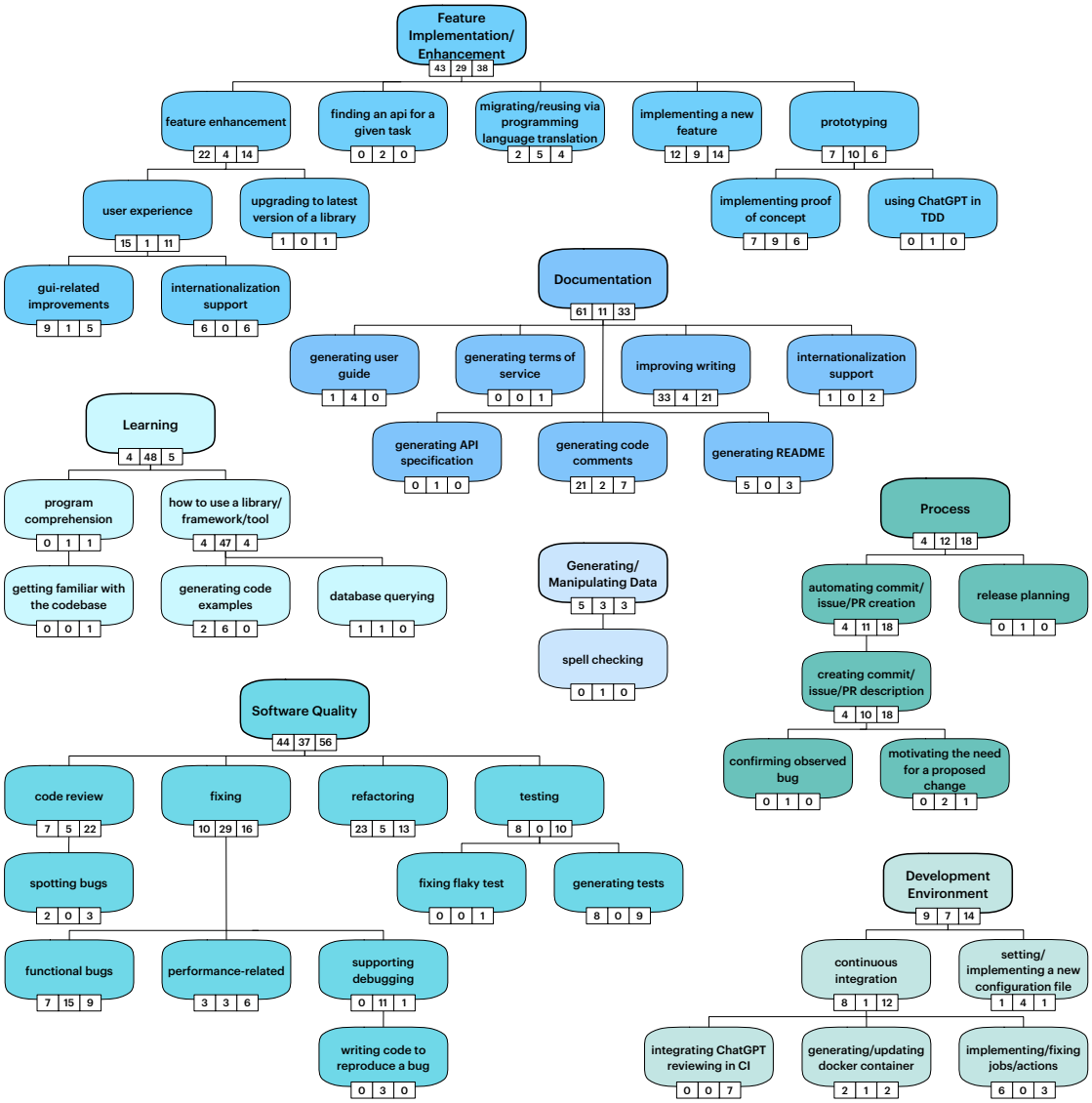


Figure D.1. Taxonomy of types of tasks automated via ChatGPT

autoupdater script commenting "*I don't know if this is the right way to do it [...] Credits to ChatGPT for the script*". While this PR has been approved and merged, it is representative of those instances in which the contributor explicitly states to be unsure about what was accomplished using ChatGPT, or even declared that they were completely unfamiliar with coding while contributing PRs or issues — see *e.g.,* [iss23a]: "*I started to edit the files with the support by ChatGPT— as said, I have no idea about coding*". 💻 Similarly to what happens when defining onboarding and contribution guidelines in open source projects [Apa, Ecl], it may be desirable to define guidelines about contributing with AI-generated code, *i.e.,* a project may decide to only welcome contributions from ChatGPT by users that are confident in assessing the correctness of the generated code.

Also, projects may need to adapt the code review process, *e.g.,* relying less on (semi-) automated code quality check (*e.g.,* a linter to check code quality) when AI-generated contributions come from users having little or no programming expertise.

💡 This finding is also relevant for research, as, for example, it may impact studies involving contributors of OSS (*e.g.,* studies on newcomers similar to those of Steinmacher *et al.* [SCTG16] or Zhou and Mockus [ZM10]). Related studies in the future should be careful about surveying developers that have only submitted AI-generated contributions, as they may not be representative of the target population of OSS developers. Clearly, the development landscape may also significantly change, as contributors mainly relying on AI when submitting code could become the norm.

⚠ In general, there is a clear risk related to the ownership and understanding of code contributed via ChatGPT, especially when it is used to contribute complete features. Such a problem has been well-summarized in a comment of a PR we inspected [pr:23c]: "*[...] I want to make something clear about code suggestions done by ChatGPT: deferring to an AI bot is not the same as code ownership [...] the idea that an author puts some code into a commit and sends it means they should have an intellectual understanding of it. PR authors should own the code they send – ownership in the sense of being able to advocate for the code*".

The consequence is that AI-generated code may require a more thorough quality assurance, but also may lead to issue triaging problems and in general maintenance issues in the absence of a real owner. Concerning the second usage scenario for prototyping (*i.e.,* using ChatGPT to draft the whole starting version of a project), a concrete example is the issue #1 from the `apple-notes-to-sqlite` project [iss23b] titled "Initial proof of concept with ChatGPT".

The issue documents the conversation between the repository's owner and ChatGPT, and resulted in the implementation of the first prototype of an application exporting Apple notes to SQLite. 💻 This project counts 120 stars on GitHub at the date of writing and is a concrete example of how ChatGPT can provide a jumpstart in software development.

Still related to prototyping, we also found one issue [iss23c] in which developers discuss the possibility of using "*ChatGPT to generate reliable code through a semi-automated Test-Driven Development (TDD) process that incorporates feedback loops*" (*using ChatGPT in TDD* category in our taxonomy). 💻 From a practitioner's perspective, this would lead to a different metaphor, in which the developer is mainly in charge of writing tests letting the LLM generate code.

💡 From a research perspective, this requires empirical investigations, as it has been done in the past for conventional TDD (*e.g.,* [BCF+21, FET+17]) thus defining a suitable process with AI in-the-loop.

Another popular sub-category is the *feature enhancement* task, in which ChatGPT is used to enhance an already existing feature (as opposed to help contributing with a new feature). This includes generic enhancements such as writing CSS for existing web pages [iss23r] or adding options to a feature [pr:23d], as well as more specific improvements that can be seen in the category's subtree. For example, we found 12 instances (6 commits + 6 PRs) in which ChatGPT has been used for internationalization purposes [pr:23e], mostly related to translating elements in the GUI, including error messages. ⚠ In some of these cases the reviewers asked whether the contributor was actually familiar with the target language or if, instead, they were just running ChatGPT and reporting the translation, with the risk of introducing internationalization issues. 💡 Such a finding is relevant for researchers working on detecting and fixing internationalization issues [EVORDO+20]. For example, ChatGPT could produce mistakes different from those typically committed by developers who manually implement internationalization.

The last sub-category we discuss is the one related to *migrating/reusing via programming language translation*, a task supported by ChatGPT in 11 of the inspected instances. 💻 Practitioners used ChatGPT to automatically translate code snippets across languages, allowing for possibilities of reuse that were unimaginable before (*e.g.,* reusing code across projects written in different languages).

An interesting example is the PR # 4559 from the `garden` project [pr:23f] in which the contributor used ChatGPT to translate from Javascript to Typescript the code of a third-party project which has not been updated in the last six years and was known to be affected by vulnerabilities. As documented by the contributor: "*I didn't find an easy fix . . . so I just created a fork of [third-party project] and asked ChatGPT to convert it to Typescript, removed the dependency on [third-party project] and later tweaked it to make sure that everything works*". 💡 Our findings confirm the relevance of research targeting the automated translation of software across programming languages [NNNN14, NNN14], but it also highlights the very strong performance of what should be considered the state-of-the-practice and, therefore, a baseline for comparing new approaches in this research thread. ChatGPT seems to be able to generalize across several languages, even by translating hundreds of lines of code, see *e.g.,* the Javascript to Python translation in the `textual-paint` project [com23a].

⚠ At the same time, recent research has pointed out perils of ML-based language translation [MZRC23], especially because the translation may not take into account that different programming languages may follow different programming paradigms (*e.g.,* object oriented *vs* functional), and the result could just be "Java with a Python syntax" or something similar. As Malyala *et al.* suggest [MZRC23], a combination of ML-based translation with static analysis and rule-based translation could be a pragmatic road to follow.

D.2.2 Process

The *process* category groups instances mentioning the usage of ChatGPT to support activities related to the development process, *e.g., release planning*, or the automation of steps needed to *create commits, PRs, issues*, *e.g.,* generating a PR description.

We found a single issue in which ChatGPT has been used to come up with ideas on how to improve a software project (*release planning* label in our taxonomy) [iss23d]. While this is a single data point and should be taken as such, we found this usage of ChatGPT extremely interesting, since it goes substantially further than what state-of-the-art tools supporting release planning are able to do. The latter usually mine data (*e.g.,* app reviews [CSPG17, PSG+15, SBR+19]) to help developers summarizing the customers' feedback and come up with aspects to improve in the software. 💻 ChatGPT does not require any sort of data mining on the developer's side and, as visible from the issue we inspected [iss23d], can be queried for general ideas about what to improve in a software or even on how to improve a specific feature/quality aspect of the software (*e.g., "Can you suggest improvements to make the help system more useful for data scientists?*", "*Suggest ways to make the help system more useful for developers*"). The contributor confirmed that "*ChatGPT came up with [...] pretty good ideas*". ⚠ One clear limitation is that requirement crowdsourcing may require up-to-date sources of information (*e.g.,* recent information about the features that competitive software implements) which ChatGPT may not have. Also, practitioners may be afraid to prompt ChatGPT with sensitive, market-competition-related information. 💡 To cope with both issues, researchers may develop approaches based on Retrieval Augmentation Generation (RAG) [LPP+20a] to combine LLMs with private or up-to-date resources.

In 32 instances developers used ChatGPT to automatically generate a commit message (*e.g.,* [com23b]) or a PR/issue description (*e.g.,* [iss23e]). The automatic generation of commit messages [DLZ+22, LXH+18, JAM17, WXL+21] and of PR title/descriptions [FZT+22, LXT+19, IZT+22] has been tackled by several researchers, especially after the wide adoption of deep learning in software engineering. While we are not aware of empirical comparisons performed between these tools and ChatGPT two observations can be made. 💡 First, the instances in our dataset show an impressive capability of ChatGPT in summarizing even complex changes spanning several files, while studies in the literature documented strong limitations of techniques for the automated generation of commit messages, mostly targeting the low-hanging fruits (*e.g., Add README*) [LXH+18]. Second, empirical comparisons should carefully consider which test instances to use, considering that the dataset on which ChatGPT has been trained is not publicly available. While this does not make it possible to ensure a lack of overlap between training and test set, an easy solution is to use very recent commits/PRs/issues as test set, since those are unlikely to have been seen by the model behind ChatGPT.

Our taxonomy also features two specializations of the *creating commit/PR/issue description* category. The first concerns a scenario in which ChatGPT has been used to confirm an observed failure as a bug: "*[failure description] asking ChatGPT suggested it to be a bug*" [iss23f]. The second represents instances in which ChatGPT was used to better motivate a proposed change. An example of this second usage scenario is the issue #1648 from the `js-libp2p` project [iss23g], being a feature request including a question posed to ChatGPT

about the usefulness of the proposed feature ("*Can't we already do this with just a libp2p stream? Why do we need HTTP?*") with the LLM providing four disadvantages of not having such a feature.

In other cases falling in this category, the contributor just describes a chat they had with ChatGPT that helped them in coming up with the issue/PR (*e.g., "this is a PR to improve the way we store thumbnails in the data folder; after a nice chat with ChatGPT I discovered why most apps do this*" [pr:23g]).

⚠ Despite the very successful applications of ChatGPT to *process*-related tasks, we observed cases of what has been recently defined as artificial hallucination [JLF+23], namely confident responses provided by an AI such as ChatGPT which look plausible to the human interacting with it but that are clearly wrong. This is reflected in negative reactions to suggestions given by ChatGPT about features to improve/implement (*e.g., "I can't really do anything about that, and it would defeat the entire purpose of [project]*" [iss23h]).

D.2.3 Learning

The *learning* category tree mostly features issues opened by users of a software project as a result of problems they are experiencing in using the related library/framework/tool. In doing so, they mention their attempt to solve the faced problem by asking ChatGPT (*e.g.,* [iss23i, iss23j]). ⚠ This is a category of task for which ChatGPT showed clear limitations related to the previously mentioned artificial hallucination issue. In 36 out of the 47 opened issues, the indications provided by ChatGPT on how to solve the problem faced by the user were wrong, resulting in negative comments either by the user itself when opening the issue (*e.g., "I even asked ChatGPT who made up some configuration options that do not exist*" [iss23j]) or by the developers replying to the issue (*e.g., "just stop asking ChatGPT about this thing, because the data that was used to train it only spans until 2021, which means everything created from 2022 (including [project]) onwards is outside of its knowledge domain; If you ask about something it doesn't know, it will make up fake answers that don't work at all, and fake libraries that don't exist*" [iss23k]). ⚠ Similarly to release planning, leveraging outdated knowledge of LLMs can be risky, therefore they need to be complemented with alternative approaches.

The *learning* tree also features two instances in which ChatGPT has been used to understand code. In these two cases, the LLM provided useful support to the developer, even in understanding code automatically generated by a framework by reverse engineering it: "*This code is very difficult for people to read because it is compiled by webpack. However, ChatGPT completed reverse-engineering in a short time*" [iss23l]. 💻 This shows the potential of ChatGPT in supporting program comprehension and on the research side 💡 suggest investigations aimed at assessing the impact of ChatGPT in program comprehension, as well as approaches to support the use of third-party LLMs on private code or other software artifacts.

D.2.4 Generating/manipulating data

Developers use ChatGPT to easily *generate/manipulate data*. The variety of data involved in this category includes strings appearing in the UI (*e.g., "add ChatGPT suggestions to bully*

messages" [com23c], "*Add extra motivational messages generated by ChatGPT*" [com23d]), fake data needed to fill templates (*e.g.,* "*add more fake data in the sample using ChatGPT*" [com23e]), or more intellectual content such as math problems for an educational project (*e.g.,* "*use ChatGPT to generate these problems and create an initial solution*" [iss23m]).

ChatGPT seems to be particularly suited in the generation of data for which correctness is not a strong requirement (*e.g.,* fake data, augmenting UI-related strings handling a dialog with the user). This makes it a suitable tool to automatically generate test inputs since even implausibly generated inputs could represent a good opportunity to assess the robustness to wrong inputs. ⚠ However, in this scenario, several risks may arise. First, LLMs can be subject to bias [CMYM20, CMM21] and may generate unwanted discriminatory or offensive text. Moreover, it cannot be excluded that LLMs could be subject to adversarial attacks, leading to the generation of unwanted outputs as it has been shown for other recommender systems [NDD+21].

D.2.5 Development environment

This tree groups together instances in which ChatGPT has been used to support and (partially) automate activities related to the *development environment*. The most popular application of ChatGPT is its *integration as reviewer in the continuous integration and delivery (CI/CD) pipeline*. In this scenario, ChatGPT is used to comment about contributed code and identify bugs and/or suboptimal implementation choices (*e.g.,* "*added bot reviewer powered with ChatGPT to help us with PR reviews*" [pr:23h]). 💻 The ChatGPT-based review is usually integrated in the continuous integration pipeline and aims at providing a first quick feedback to the contributor, without replacing (but supporting) the human reviewer. Furthermore, ChatGPT is usually combined with classic lint tools looking for issues and assessing test coverage.

Such an application confirms the relevance of the recent line of research related to the automation of code review tasks [TMM+22], for which ChatGPT could become a baseline for comparison. Future research should also consider how to properly leverage LLM-recommenders such as ChatGPT to obtain code reviews in line with an organization/project's own coding styles and guidelines. ⚠ A clear issue is the need for passing code to ChatGPT, which may not be acceptable (and even forbidden) in industrial environments. In such cases, approaches leveraging local LLMs are to be preferred.

Other tasks automated via ChatGPT concern the *implementation/fixing of jobs/actions* in continuous integration scripts and the *generation/updating of docker containers*. While ChatGPT can be a good aid to draft CI/CD scripts, this is one of those tasks in which the long training time needed for LLMs and, as a consequence, their inability to be continuously retrained to be updated, represents a strong limitation. Indeed, technologies such as GitHub actions which are used to achieve CI/CD are relatively new and rapidly evolving. This resulted in PRs contributing with CI/CD scripts created with the help of ChatGPT which, accordingly to the reviewers, were using outdated actions and commands (*e.g.,* "*the actions used are outdated*" [pr:23i]). ⚠ This highlights one strong limitation of LLMs: They might not be suitable in rapidly evolving contexts such as young technologies, programming

languages, etc.

💡 In these cases, it is possible that smaller, specialized models that can be quickly retrained might be more suitable and reliable.

D.2.6 Software quality

Software quality is the largest tree in our taxonomy in terms of number of instances (137). This indicates that developers largely leverage ChatGPT for automating tasks related to software quality improvement.

Refactoring operations recommended by ChatGPT are widely implemented by developers. This may include simple renaming [pr:23j] as well as more complex code transformations such as converting a recursive function into an iterative one: "*Thanks to ChatGPT for doing the recursion to iterative conversion*" [com23f]. 💡 While we found a wide variety of refactoring actions automated via ChatGPT, we observed a lack of code transformations involving multiple files, such as extract class refactoring. This is likely due to the limited view that ChatGPT has of the software systems, given its (current) lack of integration in the IDE. For such cases, approaches to generate suitable prompts helping ChatGPT to produce responses for more complex refactoring scenarios may be desirable.

Functional bugs have been fixed with the help of ChatGPT (31 instances in our taxonomy). ⚠ In these instances we observed two of the previously discussed issues affecting the usage of ChatGPT in open-source projects. First, ChatGPT has been used by inexperienced programmers to submit patches, possibly with little understanding of the contributed code: "*Was getting the error ... so I used ChatGPT to fix it, not sure how GitHub works ... so I gonna put it here*" [iss23n]. Second, the inability of ChatGPT to cope with complex code, similar to what we inferred looking at the automated refactorings: *e.g.,* "*Do not trust ChatGPT to fix complex code depending on multiple files; ChatGPT has no idea of the scope nor the current state of the codebase, so it will not be able to give a valid answer ...*" [iss23n].

Besides the explicit bug fixes suggested by ChatGPT, the LLM is also used to *support debugging*. 💻 This mostly comes in two fashions. The first is the expected one, with a user observing a failure in a code and asking ChatGPT what was causing it: *e.g.,* "*I asked ChatGPT what this error could be, and the AI gave me an important clue ...*" [iss23o]. The second is the usage of ChatGPT to generate a minimal, reproducible example [iss23p]. 💡 As of today, there are state-of-the-art approaches supporting the automated reproduction of bugs [BCH+23, FMB+23, ZSL+22]. This is mainly possible because such approaches are (i) tailored for specific categories of applications, *e.g.,* mobile apps, and (ii) can access the whole application code base. In the future, LLM-based approaches should be therefore able to use such information to support the automated reproduction of bugs.

The *code review* subtree collects the usage of ChatGPT as a reviewer, mostly for incoming PRs. It is worth commenting on the difference between this subtree and the previously discussed *development environment → continuous integration → integrating ChatGPT reviewing in CI*. The latter concerns instances in which developers manifested their interest in integrating ChatGPT in the CI/CD pipeline as a reviewer. The former, instead, groups instances in which the outcome of a code review performed by ChatGPT was discussed, even when

ChatGPT was not integrated into the workflow but queried through its user interface.

Similarly, the *code review → spotting bugs* differs from *support debugging* since in the former the developers were not aware of the bugs found by ChatGPT, while in the latter they used ChatGPT to help debugging after observing a failure.

Another, very relevant subtree of *software quality* is the one related to the automation of *testing* activities. While we found an instance in which ChatGPT was used to fix flaky tests [pr:23k], in the rest of cases (17 instances) the automated task is the generation of tests, *e.g.,* "*includes tests, which were coded with care by ChatGPT*" [pr:23l], "*tests were generated by ChatGPT and while it was not perfect it did a decent job at creating the unit tests code*" [pr:23m]. ⚠ The latter is only one of the comments we found in PRs which confirm the usefulness of ChatGPT as an aid to write tests rather than as a completely automated solution: *"I have generated the tests with the help of ChatGPT and manually checked all of them — it got a few of them wrong or was testing impossible cases, but it did find that one edge case*". 💡 On this line, research approaches could aim at integrating LLM-based test generation with approaches aimed at identifying and repairing broken tests [CZVO11, DGM10, PXP+22, SYM18].

D.2.7 Documentation

The last popular application of ChatGPT we discuss (105 instances in our sample) concerns the automated generation of software *documentation*. ChatGPT is used both to write documentation from scratch (38 instances) as well as to improve existing documentation (see *improving writing* category with 58 instances). In some cases, projects' users suggest to improve parts of the documentation since they found it difficult to read: *"I found the README.md a bit difficult to digest, so I utilized ChatGPT to help me simplify the content. This allowed me to better understand the library's core features and functionality. It might be worth considering a shorter, more concise version of the README for easier comprehension*" [pr:23n].

💻 It could be useful for projects' owners, especially for non-native speakers of the language used in the documentation, to consider the usage of ChatGPT to improve documentation quality.

For what concerns the generation of documentation from scratch, ChatGPT is mostly used for *commenting code*, but also for drafting *terms of service* [pr:23o], *user guides* [iss23q], and *README* files [pr:23p]. Differently from what we found for tasks related to code generation, we did not observe negative reactions of projects' owners/reviewers. This is likely due to: (i) the excellent performance of the LLM when dealing with natural language; (ii) the higher likelihood that the contributor posting ChatGPT-generated content has the actual competencies to assess whether the generated output is correct (*i.e.,* less coding skills required); (iii) the fact that, as ChatGPT generates natural language, projects' contributors see pretty obvious (and relatively straightforward) ways to improve/adapt it when necessary, and therefore there is less evidence of complaints; and (iv) the lower risk related to errors in the generation task (*e.g.,* typos *vs* bugs) except, of course, for terms-of-service. 💡 An empirical investigation aimed at studying the sentiment of reviewers when inspecting different types of AI-generated contributions (*e.g.,* code *vs* documentation) could help in better characterizing and backing up our observation. Last, but not least, also in this case, a proper (in

some cases large) prompt may be needed by ChatGPT to generate exhaustive and correct documentation.

This, in turn, may stimulate research on how to combine LLMs with software reverse engineering approaches for that purpose.

D.3 Threats to Validity

Threats to *construct validity* concern the relationship between the theory and observation. Studying the purpose of the use of ChatGPT in software development by mining software repositories has an intrinsic limitation. This is because we observe only cases where developers mention ChatGPT explicitly in a commit message, issue, or PR description. There could be other changes in which developers silently leveraged ChatGPT.

Moreover, as explained in Section D.1, we analyzed the textual content of commit messages, issues, and PRs, as they could be queried by GitHub. However, there may be other places where ChatGPT could have been mentioned, *e.g.,* code comments. These would require analyzing all projects' source code and could be considered in future work.

A further threat is due to our interpretation of ChatGPT purposes of usage, by reading and labeling commits and developers' discussions. This classification could have been affected by subjectiveness and imprecision. As explained in Section D.1, we mitigated this threat by having two annotators labeling each instance independently, and, after that, having a cooperative conflict resolution.

Threats to *internal validity* concern confounding factors internal to our study that could affect our results. During the manual analysis, we explicitly excluded cases in which the contribution of ChatGPT to a given development activity was unclear. Also, we used multiple labels where ChatGPT was used for multiple purposes.

Threats to *external validity* concern the generalizabiity of our findings. Within the construct validity threats stated above, the observed findings limit to open-source projects hosted on GitHub only.

Therefore, our study needs to be complemented by other types of studies (*e.g.,* interviews, survey questionnaires, ethnographic studies) conducted in closed-source scenarios, such as industrial environments. Moreover, we are only observing the first six months of ChatGPT usage, and it is possible that its variety of use will largely increase in the future. Last, this study is only limited to ChatGPT, and should be, in the future, extended to other general-purpose chat bots that could be used in software development, *e.g.,* including the recently-released Google Bard [Goo23]. It is possible that some of them could adopt techniques to circumvent limitations/risks we found for ChatGPT or, on the other hand, have limitations that ChatGPT does not have, including the ability to access up-to-date content.

D.4 Conclusions and Future Work

In this paper, we manually analyzed, through an open coding process, 1,501 commits, issues and PRs from open source projects in which there was documented usage of ChatGPT for the automation of software-related tasks. The goal was to categorize the type of support ChatGPT provided. The result of this analysis is a taxonomy of 45 tasks (partially) automated via ChatGPT, which we discussed highlighting ChatGPT's strengths and weaknesses and distilling implications for practitioners and researchers. The latter, together with our taxonomy, represent the main outcome of our study, and have been abstracted and summarized in Table D.1 for easier reference. Our future work will focus on validating our findings by (i) interviewing developers, and (ii) generalizing them to other general-purpose LLMs.

In addition, we plan to obtain evidence of the degree to which ChatGPT has proven beneficial for developers in the context of software-related task. Achieving this, however, requires the implementation of a meticulous study design with the explicit aim of mitigating biases and addressing significant issues that may arise during the analysis. We deliberately opted against undertaking this investigation by purely mining software repositories, as developers may be less prone to report cases in which usage attempts of ChatGPT turned out to be a failure.

Table D.1. Summary of implications for practitioners and researchers derived from our study

Insights for Practitioners

Contributions including AI-generated content

- Define guidelines for projects' contributions including AI-generated code, *e.g.,* a project may decide to only welcome AI-generated code from users that are confident in assessing the correctness of the contributed code.
- Clear risk related to the ownership and understanding of code contributed via ChatGPT, especially when it is used to contribute with a complete feature: The (human) contributor is not always able to explain or advocate for the submitted code.
- As with any AI-based solution, the usage of ChatGPT for software-related tasks may result in artificial hallucination: AI responses that look plausible to the user can be clearly wrong. The hard skills of developers remain essential in the era of AI-assisted coding.

Automation possibilities offered by ChatGPT

ChatGPT can be leveraged to support very complex tasks, for which its usage has not been documented/experimented in the literature. These include:

- Prototyping the complete first version of a project, providing a substantial jumpstart in software development.
- TDD collaboration, where the developer is mostly in charge of writing tests and delegating to LLM the code writing task.
- Translating source code across different programming languages, thus improving code reusability.
- Release planning, suggesting ideas on how to improve a software project based on what was observed in the wild.
- Data generation, *e.g.,* augmenting UI-related strings handling dialogs with the user.
- Debugging, from several different perspectives, including helping in locating the bug as well as in reproducing it.

Software-related tasks involving natural language

Due to its extensive training on natural language artifacts, ChatGPT is well-suited to support software-related tasks strongly characterized by natural language, such as the generation of software documentation.

Risks related to sensible/private information

Some of the tasks automated via ChatGPT (*e.g.,* code review) require to pass it sensible information, such as the code base itself, which may not be acceptable in industrial environments. Practitioners must carefully consider the tradeoff of using a publicly available LLM *vs* training a local LLM.

Unsuitability of ChatGPT for tasks dealing with recent technologies

ChatGPT may not be suitable for tasks requiring up-to-date technology appeared after its last retraining. LLMs leveraging up-to-date knowledge available in the wild may obtain better results.

Insights for Researchers

Implications for the design of empirical studies

- Empirical investigations studying OSS contributors may or may not consider representative developers that only submitted AI-generated code.
- ChatGPT must be considered as a baseline in works proposing novel recommenders for tasks where it was found to be useful. However, as the dataset on which ChatGPT has been trained is not publicly available, it is hard to make a fair comparison ensuring the lack of overlap between training and test set. A possible solution is to use recent data points as test set, since those are unlikely to have been seen by the model behind ChatGPT.

Studying and enhancing AI-aided development processes

Practitioners are already leveraging ChatGPT for a variety of tasks. Nevertheless, it may be useful to (empirically) devise AI-enabled development processes, with suitable guidelines. These include using ChatGPT (or similar tools):

- In TDD, with the developer being mostly in charge of writing tests and delegating to the LLM the production code.
- To support program comprehension, especially when newcomers onboard a project and must become familiar with its code base.
- To generate tests.
- To automate code review.

Moreover, researchers should focus on approaches aimed at better integrating ChatGPT or similar tools in development contexts where there is a need for:

- Understand, refactor, complete very specific code or other artifacts.
- Avoid exposing internal artifacts to the outside, *e.g.,* using Retrieval Augmentation Generation or similar approaches.
- Repair ChatGPT-generated code and tests, or adapt them in own code base.

Questioning the suitability of existing recommenders for AI-generated code

The effectiveness of recommender systems for software engineers proposed in the literature (*e.g.,* tools identifying and fixing internationalization issues, APR techniques) may need to be reassessed on AI-generated code, since the latter may have characteristics different from those of human-written code.

Bibliography

[ABBS14] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Learning natural coding conventions. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 281–293, 2014.

[ABBS15] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Suggesting accurate method and class names. In *10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE*, pages 38–49, 2015.

[ABF+19] Naveen Arivazhagan, Ankur Bapna, Orhan Firat, Dmitry Lepikhin, Melvin Johnson, Maxim Krikun, Mia Xu Chen, Yuan Cao, George F. Foster, Colin Cherry, Wolfgang Macherey, Zhifeng Chen, and Yonghui Wu. Massively multilingual neural machine translation in the wild: Findings and challenges. *CoRR*, abs/1907.05019, 2019.

[ABLL21] E. Aghajani, G. Bavota, M. Linares-Vásquez, and M. Lanza. Automated documentation of android apps. *IEEE Transactions on Software Engineering, TSE*, 47(1):204–220, 2021.

[ABLY18] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400*, 2018.

[ACB+22] Eman Abdullah AlOmar, Ben Christians, Mihal Busho, Ahmed Hamad AlKhalid, Ali Ouni, Christian Newman, and Mohamed Wiem Mkaouer. Satdbailiff-mining and tracking self-admitted technical debt. *Science of Computer Programming*, 213:102693, 2022.

[ACRC20] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. A transformer-based approach for source code summarization. *arXiv preprint arXiv:2005.00653*, 2020.

[ACRC21] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333*, 2021.

[AD22] Toufique Ahmed and Premkumar Devanbu. Few-shot training llms for project-specific code-summarization. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–5, 2022.

[agi] Utilizing fast testing to transform java development into an agile, quick release, low risk process.

[Apa] Apache Software Foundation. Guide for new project contributors `https://community.apache.org/contributors/`. Accessed: 2023-07-08.

[APDB24] Toufique Ahmed, Kunal Suresh Pai, Premkumar Devanbu, and Earl T Barr. Automatic semantic augmentation of language model prompts (for code summarization). In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, pages 1004–1004. IEEE Computer Society, 2024.

[APS16a] Miltiadis Allamanis, Hao Peng, and Charles Sutton. A convolutional attention network for extreme summarization of source code. In *International Conference on Machine Learning (ICML)*, 2016.

[APS16b] Miltiadis Allamanis, Hao Peng, and Charles A. Sutton. A convolutional attention network for extreme summarization of source code. *CoRR*, abs/1602.03001, 2016.

[ATA23] Saranya Alagarsamy, Chakkrit Tantithamthavorn, and Aldeida Aleti. A3test: Assertion-augmented automated test case generation. *arXiv preprint arXiv:2302.10352*, 2023.

[AZLY19] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

[BAY20] Shaked Brody, Uri Alon, and Eran Yahav. Neural edit completion. *arXiv preprint arXiv:2005.13209*, 2020.

[BCB14] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473, 2014.

[BCF+21] Maria Teresa Baldassarre, Danilo Caivano, Davide Fucci, Natalia Juristo, Simone Romano, Giuseppe Scanniello, and Burak Turhan. Studying test-driven development and its retainment over a six-month time span. *J. Syst. Softw.*, 176:110937, 2021.

[BCH+23] Carlos Bernal-Cárdenas, Nathan Cooper, Madeleine Havranek, Kevin Moran, Oscar Chaparro, Denys Poshyvanyk, and Andrian Marcus. Translating video recordings of complex mobile app UI gestures into replayable scenarios. *IEEE Trans. Software Eng.*, 49(4):1782–1803, 2023.

[BH95] Yoav Benjamini and Yosef Hochberg. Controlling the false discovery rate: A practical and powerful approach to multiple testing. *Journal of the Royal Statistical Society. Series B (Methodological)*, 57(1), 1995.

[BHRV21] Berkay Berabi, Jingxuan He, Veselin Raychev, and Martin Vechev. Tfix: Learning to fix coding errors with a text-to-text transformer. In *International Conference on Machine Learning*, pages 780–791. PMLR, 2021.

[BKS18] Sahil Bhatia, Pushmeet Kohli, and Rishabh Singh. Neuro-symbolic program corrector for introductory programming assignments. In *Proceedings of the 40th International Conference on Software Engineering*, pages 60–70, 2018.

[BL05] Satanjeev Banerjee and Alon Lavie. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In *Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*, pages 65–72, Ann Arbor, Michigan, June 2005. Association for Computational Linguistics.

[BLBV13] Nicolas Boulanger-Lewandowski, Yoshua Bengio, and Pascal Vincent. Audio chord recognition with recurrent neural networks. In *ISMIR*, pages 335–340. Citeseer, 2013.

[Bli] Double Blind. `https://snippets-summarization.github.io`.

[BMM09] Marcel Bruch, Martin Monperrus, and Mira Mezini. from examples to improve code completion systems. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE 2009, pages 213–222, 2009.

[BR16] Gabriele Bavota and Barbara Russo. A large-scale empirical study on self-admitted technical debt. In *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*, pages 315–326, 2016.

[bso] `https://www.crummy.com/software/BeautifulSoup/`.

[BSPC19] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. Getafix: learning to fix bugs automatically. *Proc. ACM Program. Lang.*, 3(OOPSLA):159:1–159:27, 2019.

[BVLR17] David Bingham Brown, Michael Vaughn, Ben Liblit, and Thomas Reps. The care and feeding of wild-caught mutants. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 511–522, New York, NY, USA, 2017. ACM.

[Cam18] G Ann Campbell. Cognitive complexity: An overview and evaluation. In *Proceedings of the 2018 international conference on technical debt*, pages 57–58, 2018.

[Car82] Breck Carter. On choosing identifiers. *ACM Sigplan Notices*, 17(5):54–59, 1982.

[ccn] Cc-news dataset. https://commoncrawl.org/2016/10/news-dataset-available/.

[CCP+21] Matteo Ciniselli, Nathan Cooper, Luca Pascarella, Antonio Mastropaolo, Emad Aghajani, Denys Poshyvanyk, Massimiliano Di Penta, and Gabriele Bavota. An empirical study on the usage of transformer models for code completion. *IEEE Transactions on Software Engineering*, 2021.

[ccs] Stories dataset. https://github.com/tensorflow/models/tree/archive/research/lm_commonsense#1-download-data-files.

[CDM13] Michael L Collard, Michael John Decker, and Jonathan I Maletic. srcml: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration. In *2013 IEEE International Conference on Software Maintenance*, pages 516–519. IEEE, 2013.

[CGM+13] Antonio Carzaniga, Alessandra Gorla, Andrea Mattavelli, Nicolò Perino, and Mauro Pezzè. Automatic recovery from runtime failures. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 782–791, Piscataway, NJ, USA, 2013. IEEE Press.

[cha] Chatgpt https://openai.com/blog/chatgpt.

[CHAvD21] Jeanderson Cândido, Jan Haesen, Maurício Aniche, and Arie van Deursen. An exploratory study of log placement recommendation in an enterprise system. *IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 2021.

[CHC+23] Eason Chen, Ray Huang, Han-Shin Chen, Yuen-Hsien Tseng, and Liang-Yi Li. Gptutor: a chatgpt-powered programming tool for code explanation. In *International Conference on Artificial Intelligence in Education*, pages 321–327. Springer, 2023.

[Che15] L. Chen. Continuous delivery: Huge benefits, but challenges too. *IEEE Software*, 32(2):50–54, 2015.

[Che17] Lianping Chen. Continuous delivery: Overcoming adoption challenges. *Journal of Systems and Software*, 128:72 – 86, 2017.

[CHL05] Sumit Chopra, Raia Hadsell, and Yann LeCun. Learning a similarity metric discriminatively, with application to face verification. In *2005 IEEE computer society conference on computer vision and pattern recognition (CVPR'05)*, volume 1, pages 539–546. IEEE, 2005.

[CHL+19a] Huanchao Chen, Yuan Huang, Zhiyong Liu, Xiangping Chen, Fan Zhou, and Xiaonan Luo. Automatically detecting the scopes of source code comments. *Journal of Systems and Software*, 153:45–63, 2019.

[CHL+19b] Huanchao Chen, Yuan Huang, Zhiyong Liu, Xiangping Chen, Fan Zhou, and Xiaonan Luo. Automatically detecting the scopes of source code comments. *Journal of Systems and Software, JSS*, 153:45–63, 2019.

[CJ17] Boyuan Chen and Zhen Ming Jack Jiang. Characterizing logging practices in java-based open source software projects–a replication study in apache software foundation. *Empirical Software Engineering*, 22(1):330–374, 2017.

[CKM22] Zimin Chen, Steve Kommrusch, and Martin Monperrus. Neural transfer learning for repairing security vulnerabilities in c code. *IEEE Transactions on Software Engineering*, 49(1):147–165, 2022.

[CKS+17] Alexis Conneau, Douwe Kiela, Holger Schwenk, Loic Barrault, and Antoine Bordes. Supervised learning of universal sentence representations from natural language inference data. *arXiv preprint arXiv:1705.02364*, 2017.

[CKT+19] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering*, 2019.

[CL20] Isaac Caswell and Bowen Liang. Recent advances in google translate. `https://ai.googleblog.com/2020/06/recent-advances-in-google-translate.html`, 2020.

[CLN+23] Andrew A Chien, Liuzixuan Lin, Hai Nguyen, Varsha Rao, Tristan Sharma, and Rajini Wijayawardana. Reducing the carbon impact of generative ai inference (today and in 2035). In *Proceedings of the 2nd Workshop on Sustainable Computer Systems*, pages 1–7, 2023.

[CMM21] Joymallya Chakraborty, Suvodeep Majumder, and Tim Menzies. Bias in machine learning software: why? how? what to do? In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 429–440. ACM, 2021.

[CMYM20] Joymallya Chakraborty, Suvodeep Majumder, Zhe Yu, and Tim Menzies. Fairway: a way to build fair ML software. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 654–665. ACM, 2020.

[cod] `https://github.com/jdkato/codetype`.

[Coh60] Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20(1):37–46, 1960.

[com23a] `https://github.com/1j01/textual-paint/commit/e9494ddf`, 2023.

[com23b] `https://github.com/fluxninja/aperture/commit/70a68635`, 2023.

[com23c] https://github.com/pbui/bobbit/commit/089fc145, 2023.

[com23d] https://github.com/dodona-edu/dodona/commit/9efb97f8, 2023.

[com23e] https://github.com/reorx/jsoncv/commit/1d5f8f1d, 2023.

[com23f] https://github.com/spotlightpa/almanack/commit/955fc76b, 2023.

[Con98] W. J. Conover. *Practical Nonparametric Statistics*. Wiley, 3rd edition edition, 1998.

[cop] Github copilot https://copilot.github.com.

[CPG20] A. Ciurumelea, S. Proksch, and H. C. Gall. Suggesting comment completions for python using neural language models. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 456–467, 2020.

[CR22] Heike Wehrheim Cedric Richter. Tssb-3m: Mining single statement bugs at massive scale. In *MSR*, 2022.

[CSH+19] Jinfu Chen, Weiyi Shang, Ahmed E Hassan, Yong Wang, and Jiangbin Lin. An experience report of generating load tests using log-recovered workloads at varying granularities of user behaviour. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 669–681. IEEE, 2019.

[CSPG17] Adelina Ciurumelea, Andreas Schaufelbühl, Sebastiano Panichella, and Harald C. Gall. Analyzing reviews and code of mobile apps for better release planning. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER*, pages 91–102. IEEE Computer Society, 2017.

[CSX+18] Boyuan Chen, Jian Song, Peng Xu, Xing Hu, and Zhen Ming Jiang. An automated approach to estimating code coverage measures via execution logs. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 305–316, 2018.

[CTJ+21] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[Cug] *CugLM Model*. https://github.com/LiuFang816/CugLM.

[Cun92] Ward Cunningham. The wycash portfolio management system. In *Addendum to the Proceedings on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA 1992 Addendum, Vancouver, British Columbia, Canada, October 18-22, 1992*, pages 29–30. ACM, 1992.

[CYK+18] Daniel Cer, Yinfei Yang, Sheng-yi Kong, Nan Hua, Nicole Limtiaco, Rhomni St John, Noah Constant, Mario Guajardo-Cespedes, Steve Yuan, Chris Tar, et al. Universal sentence encoder. *arXiv preprint arXiv:1803.11175*, 2018.

[CZN+22] Nathan Cassee, Fiorella Zampetti, Nicole Novielli, Alexander Serebrenik, and Massimiliano Di Penta. Self-admitted technical debt and comments' polarity: an empirical study. *Empir. Softw. Eng.*, 27(6):139, 2022.

[CZVO11] Shauvik Roy Choudhary, Dan Zhao, Husayn Versee, and Alessandro Orso. Water: Web application test repair. In *Proceedings of the First International Workshop on End-to-End Test Script Engineering*, ETSE '11, page 24–29. Association for Computing Machinery, 2011.

[DAB21] Ozren Dabic, Emad Aghajani, and Gabriele Bavota. Sampling projects in github for msr studies. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 560–564. IEEE, 2021.

[dat] `https://github.com/akoumjian/datefinder`.

[DB23] Marian Daun and Jennifer Brings. How chatgpt will change software engineering education. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*, pages 110–116, 2023.

[DCLT19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT*, pages 4171–4186, 2019.

[Dev] DEVINTA, an artificial assistant for software developers. `http://devinta.si.usi.ch/`.

[DGM10] Brett Daniel, Tihomir Gvero, and Darko Marinov. On test repair using symbolic execution. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, page 207–218. Association for Computing Machinery, 2010.

[DLD+22] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Zhiyong Wu, Baobao Chang, Xu Sun, Jingjing Xu, and Zhifang Sui. A survey on in-context learning. *arXiv preprint arXiv:2301.00234*, 2022.

[DLS22] Zishuo Ding, Heng Li, and Weiyi Shang. Logentext: automatically generating logging texts using neural machine translation. *SANER. IEEE*, 2022.

[DLZ+22] Jinhao Dong, Yiling Lou, Qihao Zhu, Zeyu Sun, Zhilin Li, Wenjie Zhang, and Dan Hao. FIRA: fine-grained graph-based code change representation for automated commit message generation. In *Proceedings of the 44th International*

Conference on Software Engineering, ICSE '22, page 970–981. Association for Computing Machinery, 2022.

[DLZS17] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1285–1298, 2017.

[DMMG22] Alexandre Decan, Tom Mens, Pooya Rostami Mazrae, and Mehdi Golzadeh. On the use of github actions in software development repositories. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 235–245. IEEE, 2022.

[DP22] Renzo Degiovanni and Mike Papadakis. μbert: Mutation testing using pre-trained language models. In *2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 160–169. IEEE, 2022.

[DRML22] Elizabeth Dinella, Gabriel Ryan, Todd Mytkowicz, and Shuvendu K Lahiri. Toga: A neural method for test oracle generation. In *Proceedings of the 44th International Conference on Software Engineering*, pages 2130–2141, 2022.

[dSAdO05] Sergio Cozzetti B. de Souza, Nicolas Anquetil, and Káthia M. de Oliveira. A study of the documentation essential to software maintenance. In *International Conference on Design of Communication*, pages 68–75, 2005.

[dSMASS17] Everton da S. Maldonado, Rabe Abdalkareem, Emad Shihab, and Alexander Serebrenik. An empirical study on the removal of self-admitted technical debt. In *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*, pages 238–248, 2017.

[dSMST17] Everton da S. Maldonado, Emad Shihab, and Nikolaos Tsantalis. Using natural language processing to automatically detect self-admitted technical debt. *IEEE Trans. Software Eng.*, 43(11):1044–1062, 2017.

[DTC+23] Zishuo Ding, Yiming Tang, Xiaoyu Cheng, Heng Li, and Weiyi Shang. Logentext-plus: Improving neural machine translation based logging texts generation with syntactic templates. *ACM Transactions on Software Engineering and Methodology*, 33(2):1–45, 2023.

[dV23] Alex de Vries. The growing energy footprint of artificial intelligence. *Joule*, 7(10):2191–2194, 2023.

[EB22] Neil A. Ernst and Gabriele Bavota. Ai-driven development is here: Should you worry? *IEEE Softw.*, 39(2):106–110, 2022.

[Ecl] Eclipse Foundation. Platform/how to contribute `https://wiki.eclipse.org/Platform/How_to_Contribute`. Accessed: 2023-07-08.

[EGOK+23] Tiago Espinha Gasiba, Kaan Oguzhan, Ibrahim Kessba, Ulrike Lechner, and Maria Pinto-Albuquerque. I'm sorry dave, i'm afraid i can't fix your code: On chatgpt, cybersecurity, and secure coding. In *4th International Computer Programming Education Conference (ICPEC 2023)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2023.

[EP22] Aryaz Eghbali and Michael Pradel. Crystalbleu: precisely and efficiently measuring the similarity of code. In *37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–12, 2022.

[EVORDO+20] Camilo Escobar-Velásquez, Michael Osorio-Riaño, Juan Dominguez-Osorio, Maria Arevalo, and Mario Linares-Vásquez. An empirical study of i18n collateral changes and bugs in guis of android apps. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 581–592, 2020.

[FA11] Gordon Fraser and Andrea 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*, ESEC/FSE '11, pages 416–419. ACM, 2011.

[FAO17] Markus Freitag and Yaser Al-Onaizan. Beam search strategies for neural machine translation. *arXiv preprint arXiv:1702.01806*, 2017.

[FCG+24] Guodong Fan, Shizhan Chen, Cuiyun Gao, Jianmao Xiao, Tao Zhang, and Zhiyong Feng. Rapid: Zero-shot domain adaptation for code search with pre-trained models. *ACM Transactions on Software Engineering and Methodology*, 2024.

[FCZ+21] Gianmarco Fucci, Nathan Cassee, Fiorella Zampetti, Nicole Novielli, Alexander Serebrenik, and Massimiliano Di Penta. Waiting around or job half-done? sentiment in self-admitted technical debt. In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021, Madrid, Spain, May 17-19, 2021*, pages 403–414, 2021.

[FET+17] Davide Fucci, Hakan Erdogmus, Burak Turhan, Markku Oivo, and Natalia Juristo. A dissection of the test-driven development process: Does it really matter to test-first or to test-last? *IEEE Trans. Software Eng.*, 43(7):597–614, 2017.

[FGM+23] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. Automated repair of programs from large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1469–1481. IEEE, 2023.

[FGT+20] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[FMB+14] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the International Conference on Automated Software Engineering*, pages 313–324, 2014.

[FMB+23] Mattia Fazzini, Kevin Moran, Carlos Bernal-Cárdenas, Tyler Wendland, Alessandro Orso, and Denys Poshyvanyk. Enhancing mobile app bug reporting via real-time understanding of reproduction steps. *IEEE Trans. Software Eng.*, 49(3):1246–1272, 2023.

[FTL+22a] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. Vulrepair: a t5-based automated software vulnerability repair. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 935–947, 2022.

[FTL+22b] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Q. Phung. Vulrepair: a T5-based automated software vulnerability repair. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, pages 935–947. ACM, 2022.

[FWG07] Beat Fluri, Michael Wursch, and Harald C Gall. Do code and comments co-evolve? on the relation between source code and comment changes. In *14th Working Conference on Reverse Engineering (WCRE 2007)*, pages 70–79. IEEE, 2007.

[FWGG09] Beat Fluri, Michael Würsch, Emanuel Giger, and Harald C. Gall. Analyzing the co-evolution of comments and source code. *Software Quality Journal*, 17(4):367–394, 2009.

[FZH+14] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. Where do developers log? an empirical study on logging practices in industry. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 24–33, 2014.

[FZT+22] Sen Fang, Tao Zhang, You-Shuai Tan, Zhou Xu, Zhi-Xin Yuan, and Ling-Ze Meng. Prhan: Automated pull request description generation based on hybrid attention network. *Journal of Systems and Software*, 185:111160, 2022.

[Gag94] Philip Gage. A new algorithm for data compression. *C Users J.*, 12(2):23?38, 1994.

[GBMZ13] Yvette Graham, Timothy Baldwin, Alistair Moffat, and Justin Zobel. Continuous measurement scales in human evaluation of machine translation. In *Proceedings of the 7th Linguistic Annotation Workshop and Interoperability with Discourse*, pages 33–41, 2013.

[GC19] Aaron Gokaslan and Vanya Cohen. Openwebtext corpus. `http://Skylion007.github.io/OpenWebTextCorpus`, 2019.

[GDM+23a] Aayush Garg, Renzo Degiovanni, Facundo Molina, Maxime Cordy, Nazareno Aguirre, Mike Papadakis, and Yves Le Traon. Enabling efficient assertion inference. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, pages 623–634. IEEE, 2023.

[GDM+23b] Aayush Garg, Renzo Degiovanni, Facundo Molina, Mike Papadakis, Nazareno Aguirre, Maxime Cordy, and Yves Le Traon. Assertion inferring mutants. *arXiv preprint arXiv:2301.12284*, 2023.

[GDX+21] Haifeng Gao, Bin Dong, Yongwei Zhangand Tianxiong Xiao, Shanshan Jiang, and Yuan Dong. An efficient method of supervised contrastive learning for natural language understanding. In *7th International Conference on Computer and Communications (ICCC)*, pages 1698–1704, 2021.

[ger] Gerrit.

[GGH+23] Shuzheng Gao, Cuiyun Gao, Yulan He, Jichuan Zeng, Lunyiu Nie, Xin Xia, and Michael Lyu. Code structure–guided transformer for source code summarization. *ACM Transactions on Software Engineering and Methodology*, 32(1):1–32, 2023.

[GHLH21] Yuxian Gu, Xu Han, Zhiyuan Liu, and Minlie Huang. Ppt: Pre-trained prompt tuning for few-shot learning. *arXiv preprint arXiv:2109.04332*, 2021.

[GHS] SEART github search. `https://seart-ghs.si.usi.ch`.

[gita] GitHub workflows. Last accessed Feb 16, 2023.

[Gitb] Github website. `https://www.github.com/`.

[GJL+16] Nentawe Gurumdimma, Arshad Jhumka, Maria Liakata, Edward Chuah, and James Browne. Crude: Combining resource usage data and error logs for accurate error detection in large-scale distributed systems. In *2016 IEEE 35th Symposium on Reliable Distributed Systems (SRDS)*, pages 51–60. IEEE, 2016.

[GK05] Robert J Grissom and John J Kim. *Effect sizes for research: A broad practical approach*. Lawrence Erlbaum Associates Publishers, 2005.

[Gol17] Yoav Goldberg. *Neural network methods in natural language processing*. Morgan & Claypool Publishers, 2017.

[Goo23] Inc. Google. Try Bard, an AI expertiment by Google `https://bard.google.com`, 2023.

[GPKS17] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. Deepfix: Fixing common c language errors by deep learning. In *Proceedings of the aaai conference on artificial intelligence*, volume 31, 2017.

[Gra12] Alex Graves. Sequence transduction with recurrent neural networks. *CoRR*, abs/1211.3711, 2012.

[GRL+21] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. Graphcodebert: Pre-training code representations with data flow. In *9th International Conference on Learning Representations, ICLR 2021*, 2021.

[GS10] Mark Gabel and Zhendong Su. A study of the uniqueness of source code. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 147–156, New York, NY, USA, 2010. ACM.

[GSDY21] David Gros, Hariharan Sezhiyan, Prem Devanbu, and Zhou Yu. Code to comment "translation": Data, metrics, baselining & evaluation. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, page 746–757, 2021.

[GWD+21] Xiang Gao, Bo Wang, Gregory J. Duck, Ruyi Ji, Yingfei Xiong, and Abhik Roychoudhury. Beyond tests: Program vulnerability repair via crash constraint extraction. *ACM Trans. Softw. Eng. Methodol.*, 30(2), 2021.

[HAMM10] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus. On the use of automated text summarization techniques for summarizing source code. In *2010 17th Working Conference on Reverse Engineering*, pages 35–44, 2010.

[Han04] John M Hancock. Jaccard distance (jaccard index, jaccard similarity coefficient). *Dictionary of Bioinformatics and Computational Biology*, 2004.

[HCL06] Raia Hadsell, Sumit Chopra, and Yann LeCun. Dimensionality reduction by learning an invariant mapping. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR*, pages 1735–1742, 2006.

[HCW+22] Xing Hu, Qiuyuan Chen, Haoye Wang, Xin Xia, David Lo, and Thomas Zimmermann. Correlating automated and human evaluation of code documentation generation quality. 31(4), 2022.

[HD17] Vincent J. Hellendoorn and Premkumar Devanbu. Are deep neural networks the best choice for modeling source code? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 763?773, 2017.

[HEBM22] Sakib Haque, Zachary Eberhart, Aakash Bansal, and Collin McMillan. Semantic similarity metrics for evaluating source code summarization. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, ICPC '22, page 36–47, 2022.

[HHC+20] Yuan Huang, Shaohao Huang, Huanchao Chen, Xiangping Chen, Zibin Zheng, Xiapu Luo, Nan Jia, Xinyu Hu, and Xiaocong Zhou. Towards automatically generating block comments for code snippets. *Information and Software Technology*, 127:106373, 2020.

[HHH+23] Hossein Hajipour, Keno Hassler, Thorsten Holz, Lea Schönherr, and Mario Fritz. Codelmsec benchmark: Systematically evaluating and finding security vulnerabilities in black-box code language models. *arXiv preprint arXiv:2302.04012*, 2023.

[HLWM20] Sakib Haque, Alexander LeClair, Lingfei Wu, and Collin McMillan. Improved automatic summarization of subroutines via attention to file context. In *MSR '20: 17th International Conference on Mining Software Repositories, 2020*, pages 300–310. ACM, 2020.

[HLX+17] Z. Han, X. Li, Z. Xing, H. Liu, and Z. Feng. Learning to predict severity of software vulnerability using only vulnerability description. In *33th IEEE International Conference on Software Maintenance and Evolution ICSME*, pages 125–136, 2017.

[HLX+18a] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension*, ICPC ?18, page 200?210. Association for Computing Machinery, 2018.

[HLX+18b] Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. Summarizing source code with transferred api knowledge. 2018.

[HLX+20a] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation with hybrid lexical and syntactical information. *Empirical Software Engineering*, 25:2179–2217, 2020.

[HLX+20b] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation with hybrid lexical and syntactical information. *Springer Empirical Software Engineering*, 25:2179–2217, 2020.

[HNT+17] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. Trade-offs in continuous integration: Assurance, security, and

flexibility. In *Proceedings of the 25th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2017*, 2017.

[Hol79] Sture Holm. A simple sequentially rejective multiple test procedure. *Scandinavian journal of statistics*, pages 65–70, 1979.

[HPGB19] Vincent J. Hellendoorn, Sebastian Proksch, Harald C. Gall, and Alberto Bacchelli. When code completion fails: a case study on real-world completions. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 960–970, 2019.

[HR18] Jeremy Howard and Sebastian Ruder. Universal language model fine-tuning for text classification. *arXiv preprint arXiv:1801.06146*, 2018.

[HRC19] Jacob Harer, Christopher P. Reale, and Peter Chin. Tree-transformer: A transformer-based method for correction of tree-structured data. *CoRR*, abs/1908.00449, 2019.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[HSN18] Hideaki Hata, Emad Shihab, and Graham Neubig. Learning to generate corrective patches using neural machine translation. *CoRR*, abs/1812.07170, 2018.

[HTZ+21] Junjie Huang, Duyu Tang, Wanjun Zhong, Shuai Lu, Linjun Shou, Ming Gong, Daxin Jiang, and Nan Duan. Whiteningbert: An easy unsupervised sentence embedding approach. In *Findings of the Association for Computational Linguistics: EMNLP*, pages 238–244, 2021.

[Hub92] Peter J Huber. Robust estimation of a location parameter. In *Breakthroughs in statistics: Methodology and distribution*, pages 492–518, 1992.

[HvH20] Wilhelm Hasselbring and André van Hoorn. Kieker: A monitoring framework for software engineering research. *Software Impacts*, 5:100019, 2020.

[HwcfCDmo20] Frank E Harrell Jr, with contributions from Charles Dupont, and many others. *Hmisc: Harrell Miscellaneous*, 2020. R package version 4.3-1.

[HWG+19] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *CoRR*, abs/1909.09436, 2019.

[HYS+22] Kai Huang, Su Yang, Hongyu Sun, Chengyi Sun, Xuejun Li, and Yuqing Zhang. Repairing security vulnerabilities using pre-trained programming language models. In *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 111–116. IEEE, 2022.

[HZD+22] Xu Han, Weilin Zhao, Ning Ding, Zhiyuan Liu, and Maosong Sun. Ptr: Prompt tuning with rules for text classification. *AI Open*, 3:182–192, 2022.

[HZW+21] Julian Harty, Haonan Zhang, Lili Wei, Luca Pascarella, Mauricio Aniche, and Weiyi Shang. Logging practices with mobile analytics: An empirical study on firebase. *Proceedings of the 2021 8th International Conference on Mobile Software Engineering and Systems (MOBILESoft-2021)*, 2021.

[ICRJ23] Ali Reza Ibrahimzada, Yang Chen, Ryan Rong, and Reyhaneh Jabbarvand. Automated bug generation in the era of large language models, 2023.

[IKCZ16] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2073–2083, Berlin, Germany, August 2016. Association for Computational Linguistics.

[ILN+23] Cristina Improta, Pietro Liguori, Roberto Natella, Bojan Cukic, and Domenico Cotroneo. Enhancing robustness of ai offensive code generators via data augmentation. *arXiv preprint arXiv:2306.05079*, 2023.

[Ima22] Saki Imai. Is github copilot a substitute for human pair-programming? an empirical study. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 319–321. IEEE, 2022.

[iss23a] `https://github.com/danielgross/whatsapp-gpt/issues/68`, 2023.

[iss23b] `https://github.com/dogsheep/apple-notes-to-sqlite/issues/1`, 2023.

[iss23c] `https://github.com/pfusik/cito/issues/80`, 2023.

[iss23d] `https://github.com/go-go-golems/glazed/issues/50`, 2023.

[iss23e] `https://github.com/talent-connect/connect/issues/658`, 2023.

[iss23f] `https://github.com/gofiber/fiber/issues/2301`, 2023.

[iss23g] `https://github.com/libp2p/js-libp2p/issues/1648`, 2023.

[iss23h] `https://github.com/pizzaboxer/bloxstrap/issues/224`, 2023.

[iss23i] `https://github.com/spring-cloud/spring-cloud-stream/issues/2643`, 2023.

[iss23j] `https://github.com/shaka-project/shaka-player/issues/5015`, 2023.

[iss23k] `https://github.com/module-federation/module-federation-examples/issues/2942`, 2023.

[iss23l] https://github.com/prosyslab-classroom/cs348-information-security/issues/365, 2023.

[iss23m] https://github.com/sirupsen/napkin-math/issues/26, 2023.

[iss23n] https://github.com/kohya-ss/sd-webui-additional-networks/issues/43, 2023.

[iss23o] https://github.com/rootzoll/raspiblitz/issues/3640, 2023.

[iss23p] https://github.com/puppeteer/puppeteer/issues/9959, 2023.

[iss23q] https://github.com/pwncollege/dojo/issues/132, 2023.

[iss23r] n. https://github.com/igrigorik/videospeed/issues/1035, 2023.

[IZT+22] Ivana Clairine Irsan, Ting Zhang, Ferdian Thung, David Lo, and Lingxiao Jiang. Autoprtitle: A tool for automatic pull request title generation. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 454–458, 2022.

[jac] *Jacoco*. https://www.eclemma.org/jacoco/.

[JAM17] S. Jiang, A. Armaly, and C. McMillan. Automatically generating commit messages from diffs using neural machine translation. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE'17, pages 135–146, October 2017. ISSN:.

[jav] *Java Parser*. https://github.com/javaparser/javaparser.

[Javnd] JavaParser. Javaparser. http://javaparser.org/, n.d.

[Jes00] Jester - the junit test tester. http://jester.sourceforge.net, 2000.

[JH10] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5):649–678, 2010.

[JJE14] René Just, Darioush Jalali, and Michael D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 437–440, New York, NY, USA, 2014. ACM.

[JLF+23] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. Survey of hallucination in natural language generation. *ACM Comput. Surv.*, 2023.

[JLL+18] Zhouyang Jia, Shanshan Li, Xiaodong Liu, Xiangke Liao, and Yunhuai Liu. Smartlog: Place error log statement by deep understanding of log intention. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 61–71. IEEE, 2018.

[JLL+23] Nan Jiang, Thibaud Lutellier, Yiling Lou, Lin Tan, Dan Goldwasser, and Xiangyu Zhang. Knod: Domain knowledge distilled tree decoder for automated program repair. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1251–1263. IEEE, 2023.

[jlo] https://marketplace.visualstudio.com/items?itemname=andreamichelezucchi.loginjector.

[JLT21] Nan Jiang, Thibaud Lutellier, and Lin Tan. Cure: Code-aware neural machine translation for automatic program repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1161–1173. IEEE, 2021.

[Jol86] Ian T. Jolliffe. *Principal Component Analysis*. Springer Series in Statistics. Springer, 1986.

[JSG+23] Harshit Joshi, José Cambronero Sanchez, Sumit Gulwani, Vu Le, Gust Verbruggen, and Ivan Radiček. Repair is nearly generation: Multilingual program repair with llms. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 5131–5140, 2023.

[JST+23] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. Inferfix: End-to-end program repair with llms. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1646–1656, 2023.

[jun] *jUnit*. `https://junit.org/junit5/`.

[KABV22] Kusum Kusum, Abrar Ahmed, C Bhuvana, and V Vivek. Unsupervised translation of programming language-a survey paper. In *2022 4th International Conference on Advances in Computing, Communication Control and Networking (ICAC3N)*, pages 384–388. IEEE, 2022.

[KCX+24] Jiaolong Kong, Mingfei Cheng, Xiaofei Xie, Shangqing Liu, Xiaoning Du, and Qi Guo. Contrastrepair: Enhancing conversation-based automated program repair via contrastive test case pairs, 2024.

[KDPT23] Ahmed Khanfir, Renzo Degiovanni, Mike Papadakis, and Yves Le Traon. Efficient mutation testing via pre-trained language models. *arXiv preprint arXiv:2301.03543*, 2023.

[KR18] Taku Kudo and John Richardson. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. *CoRR*, abs/1808.06226, 2018.

[KS19] Rafael-Michael Karampatsis and Charles A. Sutton. Maybe deep neural networks are the best choice for modeling source code. *CoRR*, abs/1903.05734, 2019.

[KS20] Rafael-Michael Karampatsis and Charles Sutton. How often do single-statement bugs occur? the manysstubs4j dataset. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 573–577, 2020.

[Kud18] Taku Kudo. Subword regularization: Improving neural network translation models with multiple subword candidates. *arXiv preprint arXiv:1804.10959*, 2018.

[Kul97] Solomon Kullback. *Information theory and statistics*. Courier Corporation, 1997.

[KW13] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[KWR10] Ninus Khamis, René Witte, and Juergen Rilling. Automatic quality assessment of source code comments: The javadocminer. In Christina J. Hopfe, Yacine Rezgui, Elisabeth Métais, Alun Preece, and Haijiang Li, editors, *Natural Language Processing and Information Systems*, pages 68–79, 2010.

[KZTC21] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. Code prediction by feeding trees to transformers. pages 150–162, 2021.

[LBM21] Alexander LeClair, Aakash Bansal, and Collin McMillan. Ensemble models for neural source code summarization of subroutines. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 286–297. IEEE, 2021.

[LCC+18] Zhiyong Liu, Huanchao Chen, Xiangping Chen, Xiaonan Luo, and Fan Zhou. Automatic detection of outdated comments during code changes. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, volume 01, pages 154–163, 2018.

[LCS20] Zhenhao Li, Tse-Hsun Chen, and Weiyi Shang. Where shall we log? studying and suggesting logging locations in code blocks. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 361–372, 2020.

[LCSH18] Heng Li, Tse-Hsun Peter Chen, Weiyi Shang, and Ahmed E Hassan. Studying software logging using topic models. *Empirical Software Engineering*, 23(5):2655–2694, 2018.

[Lev66] VI Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707, 1966.

[LGDVFW12] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 3–13. IEEE, 2012.

[LGR+21] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021.

[LH19] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *7th International Conference on Learning Representations, ICLR*, 2019.

[LHWM20] Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. Improved code summarization via a graph neural network. In *Proceedings of the 28th international conference on program comprehension*, pages 184–195, 2020.

[LHZ+24] Yichen Li, Yintong Huo, Renyi Zhong, Zhihan Jiang, Jinyang Liu, Junjie Huang, Jiazhen Gu, Pinjia He, and Michael R Lyu. Go static: Contextualized logging statement generation. *arXiv preprint arXiv:2402.12958*, 2024.

[Li20] Zhenhao Li. Towards providing automated supports to developers on writing logging statements. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, pages 198–201, 2020.

[Lin04] Chin-Yew Lin. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, pages 74–81, 2004.

[LJM19] Alexander LeClair, Siyuan Jiang, and Collin McMillan. A neural model for generating natural language summaries of program subroutines. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pages 795–806, 2019.

[LKB+18] Kui Liu, Dongsun Kim, Tegawendé F Bissyandé, Shin Yoo, and Yves Le Traon. Mining fix patterns for findbugs violations. *IEEE Transactions on Software Engineering*, 47(1):165–188, 2018.

[LLCS21] Zhenhao Li, Heng Li, Tse-Hsun Peter Chen, and Weiyi Shang. Deeplv: Suggesting log levels using ordinal based neural networks. In *2021 IEEE/ACM*

43rd International Conference on Software Engineering (ICSE), pages 1461–1472. IEEE, 2021.

[LLF+22] Xueyang Li, Shangqing Liu, Ruitao Feng, Guozhu Meng, Xiaofei Xie, Kai Chen, and Yang Liu. Transrepair: Context-aware program repair for compilation errors. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–13, 2022.

[LLG+20] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. BART: denoising for natural language generation, translation, and comprehension. In *58th Annual Meeting of the Association for Computational Linguistics, ACL*, pages 7871–7880, 2020.

[LLG+22a] Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, et al. Automating code review activities by large-scale pre-training. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1035–1047, 2022.

[LLG+22b] Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, et al. Codereviewer: Pre-training for automating code review activities. *arXiv preprint arXiv:2203.09095*, 2022.

[LLZJ20] Fang Liu, Ge Li, Yunfei Zhao, and Zhi Jin. Multi-task learning based pre-trained language model for code completion. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2020. Association for Computing Machinery, 2020.

[LM19] Alexander LeClair and Collin McMillan. Recommendations for datasets for source code summarization. *arXiv preprint arXiv:1904.02660*, 2019.

[LNB+19] Bin Lin, Csaba Nagy, Gabriele Bavota, Andrian Marcus, and Michele Lanza. On the quality of identifiers in test code. In *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 204–215. IEEE, 2019.

[LNFW12] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Trans. Software Eng.*, 38(1):54–72, 2012.

[LOG+19] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692, 2019.

[Lognd] Log4J. Apache log4j. https://logging.apache.org/log4j/2.x/, n.d.

[LOZ+21] Chen Lin, Zhichao Ouyang, Junqing Zhuang, Jianqiang Chen, Hui Li, and Rongxin Wu. Improving code summarization with block-wise abstract syntax tree splitting. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pages 184–195. IEEE, 2021.

[LPP+20a] Patrick S. H. Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems*, 2020.

[LPP+20b] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. Coconut: combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, pages 101–114, 2020.

[LRW+17] Siyang Lu, BingBing Rao, Xiang Wei, Byungchul Tak, Long Wang, and Liqiang Wang. Log-based abnormal task detection and root cause analysis for spark. In *2017 IEEE International Conference on Web Services (ICWS)*, pages 389–396. IEEE, 2017.

[LSA+20] Heng Li, Weiyi Shang, Bram Adams, Mohammed Sayagh, and Ahmed E Hassan. A qualitative study of the benefits and costs of logging from developers' perspectives. *IEEE Transactions on Software Engineering*, 2020.

[LSH17] Heng Li, Weiyi Shang, and Ahmed E Hassan. Which log level should developers choose for a new logging statement? *Empirical Software Engineering*, 22(4):1684–1716, 2017.

[LSM+17] Bin Lin, Simone Scalabrino, Andrea Mocci, Rocco Oliveto, Gabriele Bavota, and Michele Lanza. Investigating the use of code analysis and nlp to promote a consistent usage of identifiers. In *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 81–90. IEEE, 2017.

[LSZ+22] Hui Liu, Mingzhu Shen, Jiaqi Zhu, Nan Niu, Ge Li, and Lu Zhang. Deep learning based program generation from requirements text: Are we there yet? *IEEE Transactions on Software Engineering*, 48(4):1268–1289, 2022.

[LVLVP15] M. Linares-Vásquez, B. Li, C. Vendome, and D. Poshyvanyk. How do developers document database usages in source code? In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 36–41, Nov 2015.

[LWJ+22] Dongcheng Li, W Eric Wong, Mingyong Jian, Yi Geng, and Matthew Chau. Improving search-based automatic program repair with neural machine translation. *IEEE Access*, 10:51167–51175, 2022.

[LWLK17] Jian Li, Yue Wang, Michael R Lyu, and Irwin King. Code completion with neural attention and pointer networks. *arXiv preprint arXiv:1711.09573*, 2017.

[LWN20] Yi Li, Shaohua Wang, and Tien N. Nguyen. Dlfix: Context-based code transformation learning for automated program repair. New York, NY, USA, 2020. Association for Computing Machinery.

[LWN22a] Yi Li, Shaohua Wang, and Tien N Nguyen. Dear: A novel deep learning-based approach for automated program repair. In *Proceedings of the 44th international conference on software engineering*, pages 511–523, 2022.

[LWN22b] Yi Li, Shaohua Wang, and Tien N. Nguyen. Dear: A novel deep learning-based approach for automated program repair. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 511–523, New York, NY, USA, 2022. Association for Computing Machinery.

[LWZ+19] Bohong Liu, Tao Wang, Xunhui Zhang, Qiang Fan, Gang Yin, and Jinsheng Deng. A neural-network based code summarization approach by using source code and its call dependencies. In *Proceedings of the 11th Asia-Pacific Symposium on Internetware*, pages 1–10, 2019.

[LXH+18] Zhongxin Liu, Xin Xia, Ahmed E. Hassan, David Lo, Zhenchang Xing, and Xinyu Wang. Neural-machine-translation-based commit message generation: How far are we? In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 373–384, 2018.

[LXL+19] Zhongxin Liu, Xin Xia, David Lo, Zhenchang Xing, Ahmed E Hassan, and Shanping Li. Which variables should i log? *IEEE Transactions on Software Engineering*, 2019.

[LXT+19] Zhongxin Liu, Xin Xia, Christoph Treude, David Lo, and Shanping Li. Automatic generation of pull request descriptions. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 176–188, 2019.

[LXWZ24] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36, 2024.

[LYTH13] Chen Liu, Jinqiu Yang, Lin Tan, and Munawar Hafiz. R2fix: Automatically generating bug fixes from bug reports. In *2013 IEEE Sixth international conference on software testing, verification and validation*, pages 282–291. IEEE, 2013.

[LYX+20] Boao Li, Meng Yan, Xin Xia, Xing Hu, Ge Li, and David Lo. Deepcommenter: a deep code comment generation tool with hybrid lexical and syntactical information. In Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann, editors, *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 1571–1575. ACM, 2020.

[LZ18] Yuding Liang and Kenny Q. Zhu. Automatic generation of text descriptive comments for code blocks. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence and Thirtieth Innovative Applications of Artificial Intelligence Conference and Eighth AAAI Symposium on Educational Advances in Artificial Intelligence*, AAAI'18/IAAI'18/EAAI'18. AAAI Press, 2018.

[LZB+] Bin Lin, Fiorella Zampetti, Gabriele Bavota, Massimiliano Di Penta, Michele Lanza, and Rocco Oliveto. Sentiment analysis for software engineering: how far can we go? In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 94–104.

[LZJ20] Fang Liu, Lu Zhang, and Zhi Jin. Modeling programs hierarchically with stack-augmented lstm. *Journal of Systems and Software*, 164:110547, 2020.

[MAPB21] Antonio Mastropaolo, Emad Aghajani, Luca Pascarella, and Gabriele Bavota. An empirical study on code comment completion. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 159–170. IEEE, 2021.

[MAPB22] Antonio Mastropaolo, Emad Aghajani, Luca Pascarella, and Gabriele Bavota. Automated variable renaming: Are we there yet? *arXiv preprint arXiv:2212.05738*, 2022.

[mar] Github marketplace `https://github.com/marketplace?type=actions`.

[MAS+13a] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker. Automatic generation of natural language summaries for java classes. In *2013 21st International Conference on Program Comprehension (ICPC)*, pages 23–32, 2013.

[MAS+13b] Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori Pollock, and K Vijay-Shanker. Automatic generation of natural language summaries for java classes. In *2013 21st International Conference on Program Comprehension (ICPC)*, pages 23–32. IEEE, 2013.

[Mas23] Antonio Mastropaolo. Replication package, 2023. https://github.com/antonio-mastropaolo/automating-logging-acitivities.

[Mav] *Maven*. https://maven.apache.org.

[MBDP+16] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrian Marcus, and Gerardo Canfora. Arena: an approach for the automated generation of release notes. *IEEE Transactions on Software Engineering*, 43(2):106–127, 2016.

[MBP+15] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrian Marcus. How can I use this method? In *37th IEEE/ACM International Conference on Software Engineering, ICSE*, pages 880–890, 2015.

[McN47] Quinn McNemar. Note on the sampling error of the difference between correlated proportions or percentages. *Psychometrika*, 12(2):153–157, 1947.

[MCP+22] Antonio Mastropaolo, Nathan Cooper, David Nader Palacio, Simone Scalabrino, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. Using transfer learning for code-related tasks. *IEEE Transactions on Software Engineering*, 49(4):1580–1598, 2022.

[MFPB24] Antonio Mastropaolo, Valentina Ferrari, Luca Pascarella, and Gabriele Bavota. Log statements generation via deep learning: Widening the support provided to developers. *Journal of Systems and Software*, 210:111947, 2024.

[MH21a] Ehsan Mashhadi and Hadi Hemmati. Applying codebert for automated program repair of java simple bugs. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 505–509, 2021.

[MH21b] Ehsan Mashhadi and Hadi Hemmati. Applying codebert for automated program repair of java simple bugs. In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR*, pages 505–509, 2021.

[MHH+22] Alejandro Martín, Javier Huertas-Tato, Álvaro Huertas-García, Guillermo Villar-Rodríguez, and David Camacho. Facter-check: Semi-automated fact-checking through semantic similarity and natural language inference. *Knowl. Based Syst.*, 251:109265, 2022.

[MHPB11] Emerson Murphy-Hill, Chris Parnin, and Andrew P Black. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1):5–18, 2011.

[MLZ+19] Weibin Meng, Ying Liu, Yichen Zhu, Shenglin Zhang, Dan Pei, Yuqing Liu, Yihao Chen, Ruizhi Zhang, Shimin Tao, Pei Sun, et al. Loganomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs. In *IJCAI*, volume 19, pages 4739–4745, 2019.

[MM16] P. W. McBurney and C. McMillan. Automatic source code summarization of context for java methods. *IEEE Transactions on Software Engineering*, 42(2):103–119, 2016.

[MML15] Roberto Minelli, Andrea Mocci, and Michele Lanza. I know what you did last summer: an investigation of how developers spend their time. In Andrea De Lucia, Christian Bird, and Rocco Oliveto, editors, *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension, ICPC 2015, Florence/Firenze, Italy, May 16-24, 2015*, pages 25–35. IEEE Computer Society, 2015.

[MMTM12] Yuri Malheiros, Alan Moraes, Cleyton Trindade, and Silvio Meira. A source code recommender system to support newcomers. In *2012 ieee 36th annual computer xxsoftware and applications conference*, pages 19–24. IEEE, 2012.

[MOK05] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. Mujava: An automated class mutation system. *Software Testing, Verification & Reliability*, 15(2):97–133, June 2005.

[MPB22] Antonio Mastropaolo, Luca Pascarella, and Gabriele Bavota. Using deep learning to generate complete log statements. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 2279–2290. ACM, 2022.

[MRJ+19] Ali Mesbah, Andrew Rice, Emily Johnston, Nick Glorioso, and Edward Aftandilian. Deepdelta: Learning to repair compilation errors. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, pages 925–936, 2019.

[MSC+21] Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader-Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. Studying the usage of text-to-text transfer transformer to support code-related tasks. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021*, pages 336–347. IEEE, 2021.

[MSII19] Tsuyoshi Mizouchi, Kazumasa Shimari, Takashi Ishio, and Katsuro Inoue. Padla: a dynamic log level adapter using online phase detection. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 135–138. IEEE, 2019.

[MZRC23] Aniketh Malyala, Katelyn Zhou, Baishakhi Ray, and Saikat Chakraborty. On ml-based program translation: Perils and promises. In *45th International Conference on Software Engineering, ICSE '23, Companion Proceedings, 2023*, 2023.

[NCL23] CheolWon Na, YunSeok Choi, and Jee-Hyong Lee. Dip: Dead code insertion based black-box attack for programming language model. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7777–7791, 2023.

[NDD+21] Phuong T. Nguyen, Claudio Di Sipio, Juri Di Rocco, Massimiliano Di Penta, and Davide Di Ruscio. Adversarial attacks to API recommender systems: Time to wake up and smell the coffee*f*. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021*, pages 253–265, 2021.

[NLN+22] Changan Niu, Chuanyi Li, Vincent Ng, Jidong Ge, Liguo Huang, and Bin Luo. Spt-code: Sequence-to-sequence pre-training for learning source code representations. In *Proceedings of the 44th international conference on software engineering*, pages 2006–2018, 2022.

[nlt] `https://www.nltk.org`.

[NN22] Nhan Nguyen and Sarah Nadi. An empirical evaluation of github copilot's code suggestions. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, pages 1–5. IEEE, 2022.

[NNN+12] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. V. Nguyen, J. Al-Kofahi, and T. N. Nguyen. Graph-based pattern-oriented, context-sensitive source code completion. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 69–79, 2012.

[NNN+13] Hoan Anh Nguyen, Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N. Nguyen, and Hridesh Rajan. A study of repetitiveness of code changes in software evolution. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ASE'13, pages 180–190, Piscataway, NJ, USA, 2013. IEEE Press.

[NNN14] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. Migrating code with statistical machine translation. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 544–547, 2014.

[NNNN14] Anh Tuan Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. Statistical learning approach for mining API usage mappings for code migration. In *29th IEEE/ACM International Conference on Automated Software Engineering, ASE*, pages 457–468, 2014.

[NSM23] Noor Nashid, Mifta Sintaha, and Ali Mesbah. Retrieval-based prompt selection for code-related few-shot learning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2450–2462. IEEE, 2023.

[OGX12] Adam Oliner, Archana Ganapathi, and Wei Xu. Advances and challenges in log analysis. *Communications of the ACM*, 55(2):55–61, 2012.

[ON15] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks. *CoRR*, abs/1511.08458, 2015.

[Opp92] A. N. Oppenheim. *Questionnaire Design, Interviewing and Attitude Measurement*. Pinter Publishers, 1992.

[OSXJS16] Hyun Oh Song, Yu Xiang, Stefanie Jegelka, and Silvio Savarese. Deep metric learning via lifted structured feature embedding. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4004–4012, 2016.

[PAT+21] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. An empirical cybersecurity evaluation of github copilot's code contributions. *arXiv preprint arXiv:2108.09293*, 2021.

[PB17] Luca Pascarella and Alberto Bacchelli. Classifying code comments in java open-source software systems. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 227–237. IEEE, 2017.

[PDE07] Carlos Pacheco and Michael D. Ernst. Randoop: Feedback-directed random testing for java. In *OOPSLA'07*, pages 815–816, 01 2007.

[PDM12] Slav Petrov, Dipanjan Das, and Ryan McDonald. A universal part-of-speech tagset. In *Proceedings of the Eighth International Conference on Language Resources and Evaluation (LREC'12)*, pages 2089–2096, Istanbul, Turkey, May 2012. European Language Resources Association (ELRA).

[peg] *PEGASUS fine-tuned for paraphrasing*. `https://huggingface.co/tuner007/pegasus_paraphrase`.

[PFHLN22] Keyur Patel, João Faccin, Abdelwahab Hamou-Lhadj, and Ingrid Nunes. The sense of logging in the linux kernel. *Empirical Software Engineering*, 27(6):153, 2022.

[PGL+21] David Patterson, Joseph Gonzalez, Quoc Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David So, Maud Texier, and Jeff Dean. Carbon emissions and large neural network training. *arXiv preprint arXiv:2104.10350*, 2021.

[PHSP22] Lin Pan, Chung-Wei Hang, Avirup Sil, and Saloni Potdar. Improved text classification via contrastive adversarial training. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 11130–11138, 2022.

[pit] Pit - real world mutation testing `https://pitest.org`.

[PIT10] Pit. http://pitest.org/, 2010.

[Pop15] Maja Popović. chrf: character n-gram f-score for automatic mt evaluation. In *Proceedings of the tenth workshop on statistical machine translation*, pages 392–395, 2015.

[PP09] Derrin Pierret and Denys Poshyvanyk. An empirical exploration of regularities in open-source software lexicons. In *The 17th IEEE International Conference on Program Comprehension, ICPC 2009, Vancouver, British Columbia, Canada, May 17-19, 2009*, pages 228–232, 2009.

[PP21] Jibesh Patra and Michael Pradel. Semantic bug seeding: a learning-based approach for creating realistic bugs. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 906–918, 2021.

[pr:23a] https://github.com/woocommerce/woocommerce/pull/37233, 2023.

[pr:23b] https://github.com/fufexan/nix-gaming/pull/73, 2023.

[pr:23c] https://github.com/typescript-eslint/typescript-eslint/pull/6915, 2023.

[pr:23d] https://github.com/greenshot/greenshot/pull/484, 2023.

[pr:23e] https://github.com/vcmi/vcmi/pull/1659, 2023.

[pr:23f] https://github.com/garden-io/garden/pull/4553, 2023.

[pr:23g] https://github.com/spacedriveapp/spacedrive/pull/925, 2023.

[pr:23h] https://github.com/hubtype/botonic/pull/2491, 2023.

[pr:23i] https://github.com/kvas-it/pytest-console-scripts/pull/76, 2023.

[pr:23j] https://github.com/kkdai/chatgpt/pull/4, 2023.

[pr:23k] https://github.com/kiali/kiali/pull/5973, 2023.

[pr:23l] https://github.com/shilomagen/passport-extension/pull/16, 2023.

[pr:23m] https://github.com/kyverno/kyverno/pull/5834, 2023.

[pr:23n] https://github.com/failfa-st/hyv/pull/1, 2023.

[pr:23o] https://github.com/robherley/snips.sh/pull/17, 2023.

[pr:23p] https://github.com/az-digital/az_quickstart/pull/2226, 2023.

[PRWZ02] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, pages 311–318, 2002.

[PS14a] A. Potdar and E. Shihab. An exploratory study on self-admitted technical debt. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 91–100, 2014.

[PS14b] Aniket Potdar and Emad Shihab. An exploratory study on self-admitted technical debt. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, pages 91–100. IEEE Computer Society, 2014.

[PSG+15] Sebastiano Panichella, Andrea Di Sorbo, Emitza Guzman, Corrado Aaron Visaggio, Gerardo Canfora, and Harald C. Gall. How can i improve my app? classifying user reviews for software maintenance and evolution. In *IEEE International Conference on Software Maintenance and Evolution, ICSME*, pages 281–290. IEEE Computer Society, 2015.

[PSM14] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[PXP+22] Minxue Pan, Tongtong Xu, Yu Pei, Zhong Li, Tian Zhang, and Xuandong Li. Gui-guided test script repair for mobile apps. *IEEE Transactions on Software Engineering*, 48(3):910–929, 2022.

[QBO+14] Abdallah Qusef, Gabriele Bavota, Rocco Oliveto, Andrea De Lucia, and Dave Binkley. Recovering test-to-code traceability using slicing and textual analysis. *J. Syst. Softw.*, 88(C):147–168, February 2014.

[QOSnd] QOS.ch. Simple logging facade for java (slf4j). https://www.slf4j.org/, n.d.

[R C20] R Core Team. *R: A Language and Environment for Statistical Computing*, 2020.

[R+03] Juan Ramos et al. Using tf-idf to determine word relevance in document queries. In *Proc. of the 1st instructional conf. on machine learning*, volume 242, pages 29–48, 2003.

[repa] *Replication package*. `https://github.com/antonio-mastropaolo/robustness-copilot`.

[repb] Replication package. `https://github.com/antonio-mastropaolo/automatic-variable-renaming`.

[repc] Replication package https://github.com/antonio-mastropaolo/code-summarization-metric/tree/main.

[repd] Replication package https://github.com/antonio-mastropaolo/GH-WCOM.

[repe] Replication package https://github.com/antonio-mastropaolo/ICSME2021-Completion.

[repf] Replication package https://github.com/antonio-mastropaolo/SATD-Removal.

[repg] Replication package https://github.com/antonio-mastropaolo/TransferLearning4Code.

[RFA21] Devjeet Roy, Sarah Fakhoury, and Venera Arnaoudova. Reassessing automatic evaluation metrics for code summarization tasks. In *29th ACM Joint Meeting on European Software Engineering Conference and the ACM/SIGSOFT Symposium on the Foundations of Software Engineering, ESEC-FSE*, page 1105–1116, 2021.

[RG19] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084*, 2019.

[RGL+20] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis. *CoRR*, abs/2009.10297, 2020.

[RJ19] Romain Robbes and Andrea Janes. Leveraging small software engineering data sets with pre-trained neural networks. In Anita Sarma and Leonardo Murta, editors, *Proceedings of the 41st International Conference on Software Engineering: New Ideas and Emerging Results, ICSE (NIER) 2019, Montreal, QC, Canada, May 29-31, 2019*, pages 29–32. IEEE / ACM, 2019.

[RJAM17] Paige Rodeghero, Siyuan Jiang, Ameer Armaly, and Collin McMillan. Detecting user story information in developer-client conversations to generate extractive summaries. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE ?17, page 49?59, 2017.

[RRK15] M. M. Rahman, C. K. Roy, and I. Keivanloo. Recommending insightful comments for source code using crowdsourced knowledge. In *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 81–90, 2015.

[RSR+20] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits

of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, 21(1):5485–5551, 2020.

[RVY14] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 419–428, New York, NY, USA, 2014. ACM.

[RWZ10] Martin P. Robillard, Robert J. Walker, and Thomas Zimmermann. Recommendation systems for software engineering. *IEEE Softw.*, 27(4):80–86, 2010.

[RXX+19a] Xiaoxue Ren, Zhenchang Xing, Xin Xia, David Lo, Xinyu Wang, and John Grundy. Neural network-based detection of self-admitted technical debt: From performance to explainability. *ACM Trans. Softw. Eng. Methodol.*, 28(3):15, 2019.

[RXX+19b] Xiaoxue Ren, Zhenchang Xing, Xin Xia, David Lo, Xinyu Wang, and John Grundy. Neural network-based detection of self-admitted technical debt: From performance to explainability. *ACM Trans. Softw. Eng. Methodol.*, 28(3):15, 2019.

[SBR+19] Simone Scalabrino, Gabriele Bavota, Barbara Russo, Massimiliano Di Penta, and Rocco Oliveto. Listening to the crowd for the release planning of mobile apps. *IEEE Trans. Software Eng.*, 45(1):68–86, 2019.

[SBR21] Dominik Sobania, Martin Briesch, and Franz Rothlauf. Choose your programming copilot: A comparison of the program synthesis performance of github copilot and genetic programming. *arXiv preprint arXiv:2111.07875*, 2021.

[SCTG16] Igor Steinmacher, Tayana Uchôa Conte, Christoph Treude, and Marco Aurélio Gerosa. Overcoming open source project entry barriers with a portal for newcomers. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016*, pages 273–284. ACM, 2016.

[SDFS20] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1433–1443, 2020.

[SDLLR15] Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin Rinard. Automatic error elimination by horizontal code transfer across multiple applications. *SIGPLAN Not.*, 50(6):43–54, June 2015.

[SGL+20] Sean Stapleton, Yashmeet Gambhir, Alexander LeClair, Zachary Eberhart, Westley Weimer, Kevin Leach, and Yu Huang. A human study of comprehension and code summarization. In *Proceedings of the 28th International Conference on Program Comprehension*, ICPC '20, page 2–13, 2020.

[Sha15] Sina Shamshiri. Automated Unit Test Generation for Evolving Software. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, FSE'15, pages 1038–1041, Bergamo, Italy, 2015. ACM.

[She18] Alex Sherstinsky. Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network. *CoRR*, abs/1808.03314, 2018.

[SHJ13] Daniela Steidl, Benjamin Hummel, and Elmar Juergens. Quality analysis of source code comments. In *2013 21st International Conference on Program Comprehension (ICPC)*, pages 83–92, 2013.

[SKP15] Florian Schroff, Dmitry Kalenichenko, and James Philbin. Facenet: A unified embedding for face recognition and clustering. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, pages 815–823, 2015.

[SKY+19] Yusuke Shido, Yasuaki Kobayashi, Akihiro Yamamoto, Atsushi Miyamoto, and Tadayuki Matsumura. Automatic source code summarization with extended tree-lstm. In *2019 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2019.

[SLH+21] Alexey Svyatkovskiy, Sebastian Lee, Anna Hadjitofi, Maik Riechert, Juliana Vicente Franco, and Miltiadis Allamanis. Fast and memory-efficient neural code completion. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 329–340. IEEE, 2021.

[SN23] Sk Golam Saroar and Maleknaz Nayebi. Developers' perception of github actions: A survey analysis. *arXiv preprint arXiv:2303.04084*, 2023.

[SO21] Hadeel Saadany and Constantin Orasan. Bleu, meteor, bertscore: Evaluation of metrics performance in assessing critical translation errors in sentiment-oriented text. *CoRR*, abs/2109.14250, 2021.

[spa] Spacy. `https://spacy.io`.

[Spe09] Donna Spencer. *Card sorting: Designing usable categories*. Rosenfeld Media, 2009.

[Spi10] D. Spinellis. Code documentation. *IEEE Software*, 27(4):18–19, July 2010.

[SPV11a] G. Sridhara, L. Pollock, and K. Vijay-Shanker. Automatically detecting and describing high level actions within methods. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 101–110, 2011.

[SPV11b] G. Sridhara, L. Pollock, and K. Vijay-Shanker. Generating parameter comments and integrating with method summaries. In *2011 IEEE 19th International Conference on Program Comprehension*, pages 71–80, 2011.

[SPVS11] Giriprasad Sridhara, Lori Pollock, and K Vijay-Shanker. Automatically detecting and describing high level actions within methods. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 101–110. IEEE, 2011.

[src] *ScrML Website*. `https://www.srcml.org/`.

[SS20] Timo Schick and Hinrich Schütze. Exploiting cloze questions for few shot text classification and natural language inference. *arXiv preprint arXiv:2001.07676*, 2020.

[SSKM92] Mahadev Satyanarayanan, David C Steere, Masashi Kudo, and Hank Mashburn. Transparent logging as a technique for debugging complex distributed systems. In *Proceedings of the 5th workshop on ACM SIGOPS European workshop: Models and paradigms for distributed systems structuring*, pages 1–3, 1992.

[sta] *Stack Overflow*. `https://stackoverflow.com`.

[STQ+20] Kaitao Song, Xu Tan, Tao Qin, Jianfeng Lu, and Tie-Yan Liu. Mpnet: Masked and permuted pre-training for language understanding. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems*, 2020.

[SVL14] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. *CoRR*, abs/1409.3215, 2014.

[SYM18] Andrea Stocco, Rahulkrishna Yandrapally, and Ali Mesbah. Visual web test repair. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 503–514. Association for Computing Machinery, 2018.

[SZ09] David Schuler and Andreas Zeller. Javalanche: efficient mutation testing for java. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24-28, 2009*, pages 297–298, 2009.

[SZFS19] Alexey Svyatkovskiy, Ying Zhao, Shengyu Fu, and Neel Sundaresan. Pythia: Ai-assisted code completion system. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 2727–2735, 2019.

[t5-] T5 public checkpoint `gs://t5-data/pretrained_models/small`.

[Tat54] Robert F Tate. Correlation between a discrete and a continuous variable. point-biserial correlation. *The Annals of mathematical statistics*, 25(3):603–607, 1954.

[TCZ+22] Zhao Tian, Junjie Chen, Qihao Zhu, Junjie Yang, and Lingming Zhang. Learning to construct better mutation faults. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–13, 2022.

[TDS+20] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. Unit test case generation with transformers. *arXiv preprint arXiv:2009.05617*, 2020.

[TDSS22] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, and Neel Sundaresan. Generating accurate assert statements for unit test cases using pretrained transformers. pages 54–64, 2022.

[Thund] Chris Thunes. javalang. https://pypi.org/project/javalang/, n.d.

[TKD20] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. Refactoringminer 2.0. *IEEE Transactions on Software Engineering*, 2020.

[TKW+20] Michele Tufano, Jason Kimko, Shiya Wang, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, and Denys Poshyvanyk. Deepmutation: A neural mutation tool. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, pages 29–32, 2020.

[TLB+21] Ben Tang, Bin Li, Lili Bo, Xiaoxue Wu, Sicong Cao, and Xiaobing Sun. Grasp: Graph-to-sequence learning for automated program repair. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, pages 819–828. IEEE, 2021.

[TMC+21] Maria Tsimpoukelli, Jacob L Menick, Serkan Cabi, SM Eslami, Oriol Vinyals, and Felix Hill. Multimodal few-shot learning with frozen language models. *Advances in Neural Information Processing Systems*, 34:200–212, 2021.

[TMM+22] Rosalia Tufano, Simone Masiero, Antonio Mastropaolo, Luca Pascarella, Denys Poshyvanyk, and Gabriele Bavota. Using pre-trained models to boost code review automation. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 2291–2302. ACM, 2022.

[TMP+23] Rosalia Tufano, Antonio Mastropaolo, Federica Pepe, Ozren Dabić, Massimiliano Di Penta, and Gabriele Bavota. Replication package `https://github.com/unveilingchatgptsusage/unveilingchatgptsusage`, 2023.

[TMP+24] Rosalia Tufano, Antonio Mastropaolo, Federica Pepe, Ozren Dabić, Massimiliano Di Penta, and Gabriele Bavota. Unveiling chatgpt's usage in open source projects: A mining-based study, 2024.

[TMTL12] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T. Leavens. @tcomment: Testing javadoc comments to detect comment-code inconsistencies. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 260–269, 2012.

[TPB23] Rosalia Tufano, Luca Pascarella, and Gabriele Bavota. Automating code-related tasks through transformers: The impact of pre-training. In *45th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2023*, page To appear, 2023.

[TPT+21] Rosalia Tufano, Luca Pascarella, Michele Tufano, Denys Poshyvanyk, and Gabriele Bavota. Towards automating code review activities. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 163–174. IEEE, 2021.

[TPT22] Patanamon Thongtanunam, Chanathip Pornprasit, and Chakkrit Tantithamthavorn. Autotransform: Automated code transformation to support modern code review process. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pages 237–248, 2022.

[TPW+19] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. On learning meaningful code changes via neural machine translation. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 25–36, 2019.

[TR10] Andreas Thies and Christian Roth. Recommending rename refactorings. In *Proceedings of RSSE 2010 (2nd International Workshop on Recommendation Systems for Software Engineering)*, pages 1–5. ACM, 2010.

[TWB+19a] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Trans. Softw. Eng. Methodol.*, 28(4):19:1–19:29, 2019.

[TWB+19b] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. Learning how to mutate source code from bug-fixes. In *2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019, Cleveland, OH, USA, September 29 - October 4, 2019*, pages 301–312, 2019.

[Two] Reel Two. Jumble. `http://jumble.sourceforge.net`.

[TYKZ07] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /*icomment: bugs or bad comments?*/. In *21st ACM Symposium on Operating Systems Principles, SOSP*, pages 145–158, 2007.

[TYZ07] Lin Tan, Ding Yuan, and Yuanyuan Zhou. Hotcomments: How to make program comments more useful? In Galen C. Hunt, editor, *Proceedings of HotOS'07: 11th Workshop on Hot Topics in Operating Systems*. USENIX Association, 2007.

[url] `https://pypi.org/project/urlextract/`.

[VPSO23] Antonio Vitale, Valentina Piantadosi, Simone Scalabrino, and Rocco Oliveto. Using deep learning to automatically improve code readability. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 573–584. IEEE, 2023.

[VR02] W. N. Venables and B. D. Ripley. *Modern Applied Statistics with S*. Springer, New York, fourth edition, 2002. ISBN 0-387-95457-0.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[VYW+15] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar T. Devanbu, and Vladimir Filkov. Quality and productivity outcomes relating to continuous integration in github. In *ESEC/SIGSOFT FSE*, pages 805–816. ACM, 2015.

[VZG22] Priyan Vaithilingam, Tianyi Zhang, and Elena 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*, pages 1–7, 2022.

[WAN+21] Fengcai Wen, Emad Aghajani, Csaba Nagy, Michele Lanza, and Gabriele Bavota. Siri, write the next method. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE*, pages 138–149, 2021.

[WCH+22] Feng Wei, Zhenbo Chen, Zhenghong Hao, Fengxin Yang, Hua Wei, Bing Han, and Sheng Guo. Semi-supervised clustering with contrastive learning for discovering new intents. *arXiv preprint arXiv:2201.07604*, 2022.

[WCP+22] Cody Watson, Nathan Cooper, David Nader Palacio, Kevin Moran, and Denys Poshyvanyk. A systematic literature review on the use of deep learning in software engineering research. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(2):1–58, 2022.

[WFH+23] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C. Schmidt. A prompt pattern catalog to enhance prompt engineering with chatgpt, 2023.

[WGD+19] Deze Wang, Yong Guo, Wei Dong, Zhiming Wang, Haoran Liu, and Shanshan Li. Deep code-comment understanding and assessment. *IEEE Access*, 7:174200–174209, 2019.

[Wil45] Frank Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, 1945.

[WLG+23] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. Codet5+: Open code large language models for code understanding and generation, 2023.

[WLL+20] Bolin Wei, Yongmin Li, Ge Li, Xin Xia, and Zhi Jin. Retrieve and refine: exemplar-based neural comment generation. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 349–360. IEEE, 2020.

[WLT] Edmund Wong, Taiyue Liu, and Lin Tan. CloCom: Mining existing source code for automatic comment generation. In *Software Analysis, Evolution and Reengineering (SANER), 2015*, pages 380–389.

[WLX+19] Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. Code generation as a dual task of code summarization. *Advances in neural information processing systems*, 32, 2019.

[WNBL] Fengcai Wen, Csaba Nagy, Gabriele Bavota, and Michele Lanza. A large-scale empirical study on code-comment inconsistencies. In *Proceedings of the 27th International Conference on Program Comprehension, ICPC 2019*, pages 53–64.

[WPVS17] Xiaoran Wang, Lori Pollock, and K Vijay-Shanker. Automatically generating natural language descriptions for object-related statement sequences. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 205–216. IEEE, 2017.

[WRG+22] Carole-Jean Wu, Ramya Raghavendra, Udit Gupta, Bilge Acun, Newsha Ardalani, Kiwan Maeng, Gloria Chang, Fiona Aga, Jinshi Huang, Charles Bai, et al. Sustainable ai: Environmental implications, challenges and opportunities. *Proceedings of Machine Learning and Systems*, 4:795–813, 2022.

[WSD+21] Yanlin Wang, Ensheng Shi, Lun Du, Xiaodi Yang, Yuxuan Hu, Shi Han, Hongyu Zhang, and Dongmei Zhang. Cocosum: Contextual code summarization with multi-relational graph neural network. *arXiv preprint arXiv:2107.01933*, 2021.

[WSM+18] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. Glue: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*, 2018.

[WTM+20] Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. On learning meaningful assert statements for unit test cases. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 1398–1409, 2020.

[WWJH21] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics.

[WWJH23] Weishi Wang, Yue Wang, Shafiq Joty, and Steven CH Hoi. Rap-gen: Retrieval-augmented patch generation with codet5 for automatic program repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 146–158, 2023.

[WWS+22] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

[WXL+21] Haoye Wang, Xin Xia, David Lo, Qiang He, Xinyu Wang, and John Grundy. Context-aware retrieval-based deep commit message generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(4):1–30, 2021.

[WXZ23] Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. Copiloting the copilots: Fusing large language models with completion engines for automated program repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 172–184, 2023.

[WYG+22] Chaozheng Wang, Yuanhang Yang, Cuiyun Gao, Yun Peng, Hongyu Zhang, and Michael R Lyu. 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*, pages 382–394, 2022.

[WYT13] Edmund Wong, Jinqiu Yang, and Lin Tan. Autocomment: Mining question and answer sites for automatic comment generation. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 562–567. IEEE, 2013.

[WZL+20] Ruyun Wang, Hanwen Zhang, Guoliang Lu, Lei Lyu, and Chen Lyu. Fret: Functional reinforced transformer with bert for code summarization. *IEEE Access*, 8:135591–135604, 2020.

[WZS+20] Wenhua Wang, Yuqun Zhang, Yulei Sui, Yao Wan, Zhou Zhao, Jian Wu, S Yu Philip, and Guandong Xu. Reinforcement-learning-guided source code summarization using hierarchical attention. *IEEE Transactions on software Engineering*, 48(1):102–119, 2020.

[WZS+22] Wenhua Wang, Yuqun Zhang, Yulei Sui, Yao Wan, Zhou Zhao, Jian Wu, Philip S. Yu, and Guandong Xu. Reinforcement-learning-guided source code summarization using hierarchical attention. *IEEE Transactions on Software Engineering*, 48(1):102–119, 2022.

[WZY+18] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S Yu. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, pages 397–407, 2018.

[WZZX20] Wenhua Wang, Yuqun Zhang, Zhengran Zeng, and Guandong Xu. Transˆ 3: A transformer-based framework for unifying code summarization and code search. *arXiv preprint arXiv:2003.03238*, 2020.

[XBL+18] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, and S. Li. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering*, pages 951–976, 2018.

[XCZ+24] Junjielong Xu, Ziang Cui, Yuan Zhao, Xu Zhang, Shilin He, Pinjia He, Liqun Li, Yu Kang, Qingwei Lin, Yingnong Dang, et al. Unilog: Automatic logging via llm and in-context learning. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–12, 2024.

[XLD+17] Xin Xia, David Lo, Ying Ding, Jafar M. Al-Kofahi, Tien N. Nguyen, and Xinyu Wang. Improving automated bug triaging with specialized topic model. *IEEE Trans. Software Eng.*, 43(3):272–297, 2017.

[XQC+17] Yingce Xia, Tao Qin, Wei Chen, Jiang Bian, Nenghai Yu, and Tie-Yan Liu. Dual supervised learning. In *International conference on machine learning*, pages 3789–3798. PMLR, 2017.

[XWW+18] Kun Xu, Lingfei Wu, Zhiguo Wang, Yansong Feng, Michael Witbrock, and Vadim Sheinin. Graph2seq: Graph to sequence learning with attention-based neural networks. *arXiv preprint arXiv:1804.00823*, 2018.

[XWX22] Xuezheng Xu, Xudong Wang, and Jingling Xue. M3v: Multi-modal multi-view context embedding for repair operator prediction. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 266–277. IEEE, 2022.

[XYSZ21] Rui Xie, Wei Ye, Jinan Sun, and Shikun Zhang. Exploiting method names to improve code summarization: A deliberation multi-task learning approach. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pages 138–148. IEEE, 2021.

[XZ23] Chunqiu Steven Xia and Lingming Zhang. Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using chatgpt. *arXiv preprint arXiv:2304.00385*, 2023.

[YBdPS+18] Kundi Yao, Guilherme B. de Pádua, Weiyi Shang, Steve Sporea, Andrei Toma, and Sarah Sajedi. Log4perf: Suggesting logging locations for web-based systems' performance monitoring. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 127–138, 2018.

[YDY+19] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime G. Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. Xlnet: Generalized autoregressive pretraining for language understanding. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems*, pages 5754–5764, 2019.

[YL20] Michihiro Yasunaga and Percy Liang. Graph-based, self-supervised program repair from diagnostic feedback. In *International Conference on Machine Learning*, pages 10799–10808. PMLR, 2020.

[YL21] Michihiro Yasunaga and Percy Liang. Break-it-fix-it: Unsupervised learning for program repair. In *International conference on machine learning*, pages 11941–11952. PMLR, 2021.

[YLGS] Shangbo Yun, Shuhuai Lin, Xiaodong Gu, and Beijun Shen. Project-specific code summarization with in-context learning. *Available at SSRN 4705650*.

[YMX+10] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. Sherlog: error diagnosis by connecting clues from runtime logs. In *Proceedings of the fifteenth International Conference on Architectural support for programming languages and operating systems*, pages 143–154, 2010.

[YPZ12] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. Characterizing logging practices in open-source software. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 102–112. IEEE, 2012.

[YSHL22] Zhou Yang, Jieke Shi, Junda He, and David Lo. Natural attack for pre-trained models of code. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1482–1493, 2022.

[YWCS18] Ziyu Yao, Daniel S Weld, Wei-Peng Chen, and Huan Sun. Staqc: A systematically mined question-code dataset from stack overflow. In *Proceedings of the 2018 World Wide Web Conference*, pages 1693–1703, 2018.

[YXF+23] Wenhao Ye, Jun Xia, Shuo Feng, Xiangyu Zhong, Shuai Yuan, and Zhitao Guan. Fixgpt: A novel three-tier deep learning model for automated program repair. In *2023 8th International Conference on Data Science in Cyberspace (DSC)*, pages 499–505. IEEE, 2023.

[YXZ+20] Wei Ye, Rui Xie, Jinglei Zhang, Tianxiang Hu, Xiaoyin Wang, and Shikun Zhang. Leveraging code generation to improve code retrieval and summarization via dual learning. In *Proceedings of The Web Conference 2020*, pages 2309–2319, 2020.

[YZH+22] Wei Yuan, Quanjun Zhang, Tieke He, Chunrong Fang, Nguyen Quoc Viet Hung, Xiaodong Hao, and Hongzhi Yin. Circle: continual repair across programming languages. In *Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis*, pages 678–690, 2022.

[YZP+12] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. Improving software diagnosability via log enhancement. *ACM Transactions on Computer Systems (TOCS)*, 30(1):1–28, 2012.

[ZAX+22] Shuyan Zhou, Uri Alon, Frank F Xu, Zhengbao Jiang, and Graham Neubig. Docprompting: Generating code by retrieving the docs. In *The Eleventh International Conference on Learning Representations*, 2022.

[ZCG+22] Jialu Zhang, José Cambronero, Sumit Gulwani, Vu Le, Ruzica Piskac, Gustavo Soares, and Gust Verbruggen. Repairing bugs in python assignments using large language models. *arXiv preprint arXiv:2209.14876*, 2022.

[ZCH+20] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *AI open*, 1:57–81, 2020.

[ZCSC19] Yi Zeng, Jinfu Chen, Weiyi Shang, and Tse-Hsun Peter Chen. Studying the characteristics of logging practices in mobile apps: a case study on f-droid. *Empirical Software Engineering*, 24(6):3394–3434, 2019.

[ZFSD21] Fiorella Zampetti, Gianmarco Fucci, Alexander Serebrenik, and Massimiliano Di Penta. Self-admitted technical debt practices: a comparison between industry and open-source. *Empir. Softw. Eng.*, 26(6):131, 2021.

[ZHCHL20] Rui Zhou, Mohammad Hamdaqa, Haipeng Cai, and Abdelwahab Hamou-Lhadj. Mobilogleak: a preliminary study on data leakage caused by poor logging practices. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 577–581. IEEE, 2020.

[ZHF+15] Jieming Zhu, Pinjia He, Qiang Fu, Hongyu Zhang, Michael R Lyu, and Dongmei Zhang. Learning to log: Helping developers make informed logging decisions. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 415–425. IEEE, 2015.

[ZJW+23] Yuwei Zhang, Zhi Jin, Zejun Wang, Ying Xing, and Ge Li. Saga: Summarization-guided assert statement generation. *arXiv preprint arXiv:2305.14808*, 2023.

[ZKL+22] Albert Ziegler, Eirini Kalliamvakou, X Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. Productivity assessment of neural code completion. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pages 21–29, 2022.

[ZKW+19] Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q Weinberger, and Yoav Artzi. Bertscore: Evaluating text generation with bert. *arXiv preprint arXiv:1904.09675*, 2019.

[ZKW+20] Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q. Weinberger, and Yoav Artzi. Bertscore: Evaluating text generation with BERT. In *8th International Conference on Learning Representations, ICLR*, 2020.

[ZKZ+15] Yukun Zhu, Ryan Kiros, Richard S. Zemel, Ruslan Salakhutdinov, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. In *2015 IEEE International Conference on Computer Vision, ICCV*, pages 19–27, 2015.

[ZLJX23] Yuwei Zhang, Ge Li, Zhi Jin, and Ying Xing. Neural program repair with program dependence analysis and effective filter mechanism. *arXiv preprint arXiv:2305.09315*, 2023.

[ZM10] Minghui Zhou and Audris Mockus. Growth of newcomer competence: challenges of globalization. In *Proceedings of the Workshop on Future of Software Engineering Research, FoSER 2010, at the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 443–448. ACM, 2010.

[ZNA+17] Fiorella Zampetti, Cedric Noiseux, Giuliano Antoniol, Foutse Khomh, and Massimiliano Di Penta. Recommending when design technical debt should be self-admitted. In *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*, pages 216–226, 2017.

[ZNDP22] Fiorella Zampetti, Vittoria Nardone, and Massimiliano Di Penta. Problems and solutions in applying continuous integration and delivery to 20 open-source cyber-physical systems. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 646–657, 2022.

[ZPN+22a] Jiyang Zhang, Sheena Panthaplackel, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. Coditt5: Pretraining for source code and natural language editing.

In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–12, 2022.

[ZPN+22b] Jiyang Zhang, Sheena Panthaplackel, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. CoditT5: pretraining for source code and natural language editing. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*, pages 22:1–22:12. ACM, 2022.

[ZPX+19] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Dewei Liu, Qilin Xiang, and Chuan He. Latent error prediction and fault localization for microservice applications by learning from system trace logs. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 683–694, 2019.

[ZSD18] Fiorella Zampetti, Alexander Serebrenik, and Massimiliano Di Penta. Was self-admitted technical debt removal a real removal?: an in-depth perspective. In *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, pages 526–536. ACM, 2018.

[ZSD20] Fiorella Zampetti, Alexander Serebrenik, and Massimiliano Di Penta. Automatically learning patterns for self-admitted technical debt removal. In *27th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2020, London, ON, Canada, February 18-21, 2020*, pages 355–366. IEEE, 2020.

[ZSL+22] Yu Zhao, Ting Su, Yang Liu, Wei Zheng, Xiaoxue Wu, Ramakanth Kavuluru, William G. J. Halfond, and Tingting Yu. Recdroid+: Automated end-to-end crash reproduction from bug reports for android apps. *ACM Trans. Softw. Eng. Methodol.*, 31(3):36:1–36:33, 2022.

[ZSX+21] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. A syntax-guided edit decoder for neural program repair. In *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, pages 341–353, 2021.

[ZSZ+23] Qihao Zhu, Zeyu Sun, Wenjie Zhang, Yingfei Xiong, and Lu Zhang. Tare: Type-aware neural program repair. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1443–1455. IEEE, 2023.

[ZVP+20] Fiorella Zampetti, Carmine Vassallo, Sebastiano Panichella, Gerardo Canfora, Harald Gall, and Massimiliano Di Penta. An empirical characterization of bad practices in continuous integration. *Empirical Software Engineering*, 25:1095–1135, 2020.

[ZWZ+20a] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. Retrieval-based neural source code summarization. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 1385–1397, 2020.

[ZWZ+20b] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. Retrieval-based neural source code summarization. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 1385–1397. IEEE, 2020.

[ZXL+19] Xu Zhang, Yong Xu, Qingwei Lin, Bo Qiao, Hongyu Zhang, Yingnong Dang, Chunyu Xie, Xinsheng Yang, Qian Cheng, Ze Li, et al. Robust log-based anomaly detection on unstable log data. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 807–817, 2019.

[ZYD+19] Chen Zhi, Jianwei Yin, Shuiguang Deng, Maoxin Ye, Min Fu, and Tao Xie. An exploratory study of logging configuration practice in java. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 459–469. IEEE, 2019.

[ZYY+19] Yu Zhou, Xin Yan, Wenhua Yang, Taolue Chen, and Zhiqiu Huang. Augmenting java method comments generation with context information based on neural networks. *Journal of Systems and Software*, 156:328–340, 2019.

[ZZLW17] Wenhao Zheng, Hong-Yu Zhou, Ming Li, and Jianxin Wu. Code attention: Translating code to comments by exploiting domain features. *arXiv preprint arXiv:1709.07642*, 2017.

[ZZSL20a] Jingqing Zhang, Yao Zhao, Mohammad Saleh, and Peter Liu. Pegasus: Pre-training with extracted gap-sentences for abstractive summarization. In *International Conference on Machine Learning*, pages 11328–11339. PMLR, 2020.

[ZZSL20b] Jingqing Zhang, Yao Zhao, Mohammad Saleh, and Peter J. Liu. PEGASUS: pre-training with extracted gap-sentences for abstractive summarization. In *37th International Conference on Machine Learning, ICML*, pages 11328–11339, 2020.