Università della Svizzera italiana

Software Institute

Automating Code Review

Rosalia Tufano

Research Advisor

Prof. Dr. Gabriele Bavota

SEART

Dissertation Committee

Prof. Carlo Alberto Furia	Università della Svizzera italiana, Switzerland
Prof. Paolo Tonella	Università della Svizzera italiana, Switzerland
Prof. Massimiliano Di Penta	Università degli Studi del Sannio
Prof. Sonia Haiduc	Florida State University
Prof. Alexander Serebrenik	Eindhoven University of Technology

Dissertation accepted on 24 November 2023

Research Advisor
Prof. Gabriele Bavota

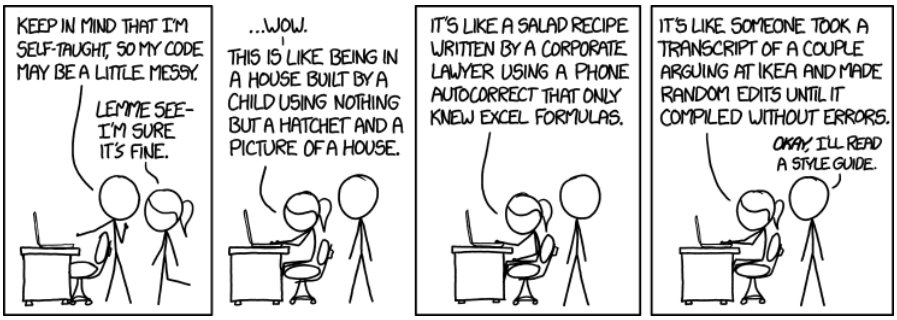
Ph.D. Program Co-Director
Prof. Walter Binder

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.

Rosalia Tufano
Lugano, 24 November 2023

Abstract

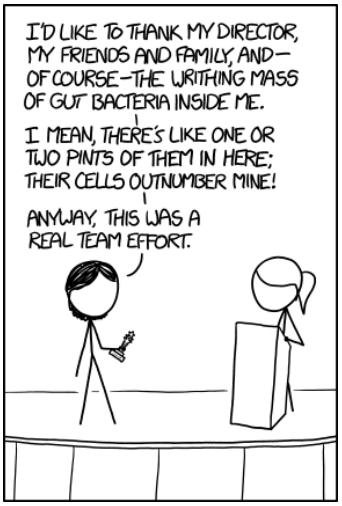


Several empirical studies have provided evidence that low code quality is generally associated with higher fault-proneness, lower productivity, more rework, and more effort for developers. In response to the need for improving code quality, code review has been widely adopted in open source and industrial projects. While empirical studies showed the undoubted advantages of code review (*e.g.,* less buggy code) its cost is non-negligible. Indeed, code review requires several developers to be allocated, with different roles, on the same coding activity. In particular, a *contributor* submits a code change to be reviewed and one or more *reviewers* inspect the change, comment on it (*e.g.,* by providing recommendations for improvement), and judge whether it is of sufficient quality to be merged.

The goal of our research is to (partially) automate this time-consuming process. The final goal is not to replace developers during code reviews but work with them in tandem by automatically solving (or identifying) code quality issues that developers would manually catch and fix. In particular, we propose techniques exploiting deep learning models to automate three code review tasks related to (i) commenting in natural language a code change submitted for review with the goal of recommending how to improve its quality as a human reviewer would do; and (ii) automatically implementing code changes usually required in the code review process.

We empirically evaluated the proposed solutions both quantitatively and qualitatively, disclosing their strengths and weaknesses. Despite the identified limitations, we show the promise of automating code review tasks using deep learning, pointing to directions for future work in the area.

Acknowledgments



The research presented in this thesis would not have been the same without all the members of the team in which I worked, and this is why my first big thank you goes to them, to the magnificent SEART group. To those who started the journey with me, with whom I shared most of the days and a 12-hour trip in Business for free. But also to those who were about to end when I started, or those who arrived later, although we didn't spend much time together, they managed to leave something of themselves in my journey.

Behind a great group there is always a great leader, and ours is no exception. Gabriele is the most atypical advisor I have met. You know all the memes about advisors that scare you and that populate PhD students' nightmares, nothing further. In 4 years he has taught me so much and, using the words of Bilbo Beggins, 4 *years among such extraordinary and admirable people are not enough!* And in fact, who's going to get me out of here? I think I still have a lot to learn from him and, even if it sounds a bit like a threat, I will take advantage of it until the end! [1]

The hardest thing to manage during a PhD? Social life. For my doctorate I moved to a new city, a new country and it wasn't easy. But luckily the friends you can call friends don't disappear as the distance increases or the time spent together decreases. Even though we are now scattered across the world, the common thread that connects us is stronger than ever. I'm talking about Enzo and Valerio, who with their Sunday "morning" video calls helped me not to go crazy in the most difficult moments. The crazy weekends traveling between Barcelona and Naples were the best and helped me understand that you don't have to go too far to find happiness! [2]

Obviously friends can also be found in Lugano, and I also have to thank the new ones I met here for having had a semblance of an active social life in these doctoral years. Thanks to the *Chicas* for the chill moments but also for the slightly crazy ones. And thanks to someone special who kept me company during sleepless nights [3], prepared food for me when I didn't have time and simply was with me in silence when I had too much work to finish at home.

[1] Yes, I will also continue the post-doc with him.

[2] Until one of your best friends goes off to seek his fortune in Australia. Thanks Valerio! :)

[3] Or should I curse them for not letting me get enough sleep?

All the people named so far have given me so much, but I literally wouldn't be here without my family. My parents always supported me and did everything they could to make sure I had a bright future no matter what direction I took. My brother, this mythological being that the entire SE research community knows, inspired me and gave me the best advice when I needed it. Even though our relationship has never been like those of brothers and sisters you see in Netflix TV series, even if we reply to each other's messages after days (if we reply), we have an ICSE paper together, and a paper is forever!

I would also like to sincerely thank the members of my dissertation committee: Prof. Carlo Alberto Furia, Prof. Paolo Tonella, Prof. Alexander Serebrenik, Prof. Sonia Haiduc and Prof. Massimiliano Di Penta. A big thank you for reading this thesis and taking part to my defense.

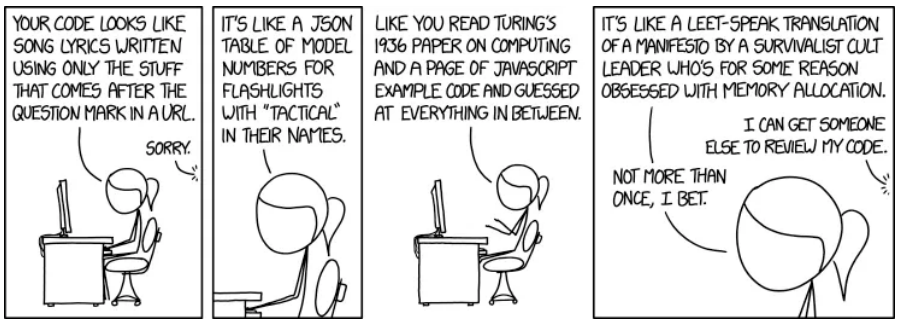
Last but not least, the biggest thank you of all obviously goes to my cat: Gandalfino. He was with me at the beginning of this journey, when I moved to Lugano way back in February 2020 he was with me and we faced Covid together! For his happiness I had to bring him back home to Italy but if it hadn't been for Gandalfino the months of quarantine and isolation would have been a mental prison. I know that many do not believe it is possible for cats to become attached to humans like dogs, but it happens to them too and Gandalfino is proof of this. The meows from behind the door when he heard me coming, the cuddles and the stolen warmth in winter, the alarm with the bites at 4 in the morning, helped make me feel at home in a place where I didn't know anyone (yet).

Oh right, thanks also to those who will actually read this thesis!

Contents

1

Introduction



Code Review is the process of analyzing source code written by a teammate to judge whether it is of sufficient quality to be integrated into the main code trunk. Recent studies provided evidence that reviewed code has lower chances of being buggy [MKAH14, MMK15, BR15] and exhibits higher internal quality [BR15], likely being easier to comprehend and maintain. Given these benefits, code reviews are widely adopted both in industrial and open source projects with the goal of finding defects, improving code quality, and identifying alternative solutions.

The benefits brought by code reviews do not come for free. Indeed, code reviews add additional expenses to the standard development costs due to the allocation of one or more reviewers having the responsibility of verifying the correctness, quality, and soundness of newly developed code. Bosu and Carver report that developers spend, on average, more than six hours per week reviewing code [BC13]. This is not surprising considering the high number of code changes reviewed in some projects: Rigby and Bird [RB13] show that industrial projects, such as Microsoft Bing, can undergo thousands of code reviews per month (∼3k in the case of Bing). Also, as highlighted by Czerwonka *et al.* [CGT15], the effort spent in code review does not only represent a cost in terms of time, but also pushes developers to switch context from their tasks.

The high cost of code review makes it an ideal candidate for solutions targeting its (partial) automation via recommender systems. In the context of software engineering (SE), recommender systems have been defined by Robillard *et al.* as "*software tools that can assist developers with a wide range of activities, from reusing code to writing effective bug reports*" [RWZ09].

When the research documented in this thesis started, most of the code review automation techniques focused on tasks which can be tackled with a machine learning (ML) classifier. For example, several researchers proposed techniques recommending the best suited reviewer for a change at hand [Bal13, TTK+15, XLWY15, OKI16, YCLW16, YWYW16, ZKB16, XSJ+17, JYH+17, FPS18, AKB+19, LWW+19, STD19, JLZ+19, MR20, ATD+20, SGBU20, COM+21, TTDE21, PT22], while others focused on predicting whether a code change submitted for review will be merged [WLZ22, WZ22, LLG+22, SLL+19, FXLL18, IAS+22], or on classifying the sentiment in reviewers' comments [ABIR17, EMHK+20]. However, the recent rise of generative deep learning (DL) models in SE made it possible automating more challenging tasks, requiring the generation of textual content, including source code.

The successful application of DL models to SE is in part due to the unprecedented amount of software-related data that can be found in open source projects hosted on platforms such as GitHub. At the time of writing GitHub counts over 100 million users[1] who submitted 413 million open source contributions only in 2022[2]. Looking at the whole history of contributions, it is safe to estimate billions of code contributions available on such a platform. Taking advantage of this data, DL-based models have been trained to support several SE tasks, such as bug-fixing [DMP+17, LNNN15, LHL+17], code smell detection [LXH+18], source code generation [CSM+18, GZZK16, HD17, KS19, Whi15, WVVP15], software testing [CPML18, GPS17, SDR+18] and program repair [BKS18, HOL+18, LKB+18, TWB+19, WSS17]. For example, a DL model has been trained to fix real bugs in Java code as developers would do [TWB+19]. To this aim, the authors collected millions of bug-fixing commits (*i.e.,* code changes aimed at fixing a bug) from GitHub, and trained the model to learn the code transformations needed to convert a buggy piece of code (*i.e.,* the one before the bug-fixing commit) into a fixed (correct) code (*i.e.,* the one after the bug-fixing commit).

Among the DL models used in the literature, pre-trained transformers [VSP+17] have shown impressive capabilities in adapting to any sort of task which can be formulated in a text-to-text fashion (*i.e.,* both the input and the output of the model is represented as a stream of textual tokens) [RSR+20]. These models are pre-trained in an unsupervised way on a dataset featuring instances written in a language of interest (*e.g.,* English), with the goal of acquiring knowledge about the statistical distribution of tokens in the language. An example of pre-training task, is the *masked language modeling*, in which 15% of tokens in an instance (*e.g.,* an English sentence) are masked with the model in charge of predicting them. Once pre-trained, the model is then fine-tuned (usually in a supervised way) for the task of interest (*e.g.,* translating between English and other languages).

[1] https://github.blog/2023-01-25-100-million-developers-and-counting/
[2] https://octoverse.github.com

Given the availability of powerful generative DL models, the vast amount of open source data present in forges such as GitHub, and the abundant evidence about successful applications of DL to SE tasks, the automation of code review is strongly under-exploited.

1.1 Thesis Statement

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

> *Generative DL models can be trained to imitate human developers involved in the code review process, thus enabling a whole new level of automation for code review tasks.*

To validate our thesis, we investigate the possibility to exploit DL-based solutions in the automation of three code-review tasks. The first, named *code-to-code*, provides as input to the DL model the code submitted for review and tries to predict (and generate) the code output of the review process (*i.e.,* a revised version of the submitted code implementing changes likely to be required during code review). This task can be used by the contributor to get a first automated feedback about the implemented change, even before starting the code review process.

The second, named *code & comment-to-code*, provides the model with the code submitted for review and a natural language comment written by the reviewer to ask for specific code changes. In this case, the DL model is expected to automatically revise the code in such a way to address the reviewer's comment. This task can be useful to the reviewer, to provide the contributor with an example of how to implement a requested change, and to the contributor, to get a recommendation on how to address the reviewer's comment.

The third task we tackled is *code-to-comment*, in which the model takes as input the code submitted for review and it is expected to generate natural language comments asking for improvements as a human reviewer would do.

We assess the proposed solutions both quantitatively and qualitatively, pointing to their strengths and limitations and concluding the thesis with a research roadmap on code review automation.

1.2 Research Contributions

The contributions of this thesis can be grouped in three high-level categories: (i) the proposal of techniques to support developers in the code review process (Chapters 3 and 4); (ii) the study of the impact of the pre-training phase on the performance of models to automate code review and, more in general, code-related tasks (Chapter 5); and (iii) an empirical investigation about the strengths and weaknesses of state-of-the-art techniques for code review automation (Chapter 6).

The research presented in this thesis is conducted in the context of the DEVINTA ERC project [Dev].

1.2.1 Deep Learning for Code Review Automation

We defined three tasks that can be automated to support developers during the code review process: *code-to-code*, *code & comment-to-code* and *code-to-comment*. Then, we:

1. Built two code review datasets featuring data about code review activities mined from open source projects. The two datasets feature triplets reporting: (i) the code submitted for review, (ii) reviewers' comments in natural language suggesting code changes aimed at improving the quality of the submitted code; and (iii) the revised version of the submitted code implementing the reviewers' recommendations.

2. Proposed different solutions to automate the defined tasks. First, we experiment with classic encoder-decoder models. A "vanilla" transformer has been used to automate the *code-to-code* task, with the model taking as input the code under review and generating its revised version. A modified version of the transformer featuring an extra encoder (for a total of two encoders) has then been used to automate the *code & comment-to-code* task, with the model able to take as input also a reviewer's comment to address besides the code submitted for review. Finally, a large pre-trained model has been used to automate all three targeted tasks, achieving better performance as compared to the previous models.

3. Empirically evaluated and compared the proposed techniques.

This part of the thesis resulted in the following publications [TPT+21, TMM+22]:

> **Towards Automating Code Review Activities**
> Rosalia Tufano, Luca Pascarella, Michele Tufano, Denys Poshyvanyk, Gabriele Bavota. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE 2021)*, pp. 163-174.

> **Using Pre-Trained Models to Boost Code Review Automation**
> Rosalia Tufano, Simone Masiero, Antonio Mastropaolo, Luca Pascarella, Denys Poshyvanyk, Gabriele Bavota. In *Proceedings of the 44th International Conference on Software Engineering (ICSE 2022)*, pp. 2291-2302

1.2.2 Studying the Impact of the Pre-training on the Automation of Code-related Tasks

We studied the extent to which different pre-training strategies may contribute in improving the automation of code-related tasks, including the code review ones targeted in this thesis. In particular, we:

- Ran a systematic literature review to identify the most used pre-training objectives to automate code-related tasks in SE.

- Experimented how the performance of pre-trained models change when adopting the three most popular pre-training objectives used in the SE literature as well as pre-training objectives tailored for the specific downstream task at hand.

The experiment has been conducted on five tasks: *code & comment-to-code*, *code-to-comment*, *bug-fixing*, *code summarization*, and *code completion*.

- Provided clear guidelines on the best pre-training strategy to adopt when dealing with code-related tasks.

This study resulted in the following publication [TPB23]:

> **Automating Code-Related Tasks Through Transformers: The Impact of Pre-training**
> Rosalia Tufano, Luca Pascarella, Gabriele Bavota. In *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023)*, pp. 2425-2437

1.2.3 Investigating Strengths and Weaknesses of the State-of-the-art

Several techniques have been built on top of our proposal to automate code review tasks [TPT22a, HHH+22, LLG+22, LYJ+22, HTTA22]. These approaches, including ours, have been mostly evaluated using quantitative metrics (*e.g.,* the percentage of correct predictions). We run a more qualitative investigation of what the strengths and the weaknesses of these techniques are. In particular, we:

- Manually inspected a statistically significant sample of correct and wrong predictions generated by three state-of-the-art techniques [TMM+22, HTTA22, LLG+22] for the automation of the *code & comment-to-code* and the *code-to-comment* tasks. The goal of the manual inspection was to characterize the scenarios in which these techniques tend to succeed and fail.
- Built two taxonomies (one for the *code & comment-to-code* task, the other one for *code-to-comment* task) showing which types of code changes are well supported by the considered approaches and which not (*e.g.,* which they can automatically implement in the context of the *code & comment-to-code* task).
- Identified problematic instances in the inspected datasets, calling for higher-quality datasets to be used in the future.
- Compared the performance of state-of-the-art approaches for the automation of code review with a that of a large language model, namely ChatGPT.

Our investigation resulted in the following article currently under review:

> **Code Review Automation: Strengths and Weaknesses of the State of the Art**
> Rosalia Tufano, Ozren Dabić, Antonio Mastropaolo, Matteo Ciniselli, Gabriele Bavota. *Under review at IEEE Transactions on Software Engineering (TSE) after Major Revisions required*

1.3 Outline

This thesis is structured in the following chapters:

Chapter 2 presents an overview of the state-of-the-art regarding code review automation. It includes the methodology used to conduct a SLR to identify the relevant studies, their discussion, and a classification of the tasks automated in the literature.

Chapter 3 describes our fist attempt to automate the *code-to-code* and *code & comment-to-code* tasks. It includes the description of the tasks, the process adopted to build the used dataset, the customization we made to the Transformer models and their evaluation.

Chapter 4 reports our effort to overcome the limitations of our first approach. It includes the description of the new approach based on pre-trained models, the introduction of a new task (*code-to-comment*), the building of a new larger dataset and the evaluation of the trained models.

Chapter 5 investigates the impact of the pre-training in the context of code-related tasks, including the code review ones subject of this thesis. It includes a SLR to identify common pre-training objectives used in SE and experiments to evaluate which of them is better suited for the automated tasks.

Chapter 6 presents an overview of strengths and weaknesses of the state-of-the-art approaches proposed to automate two of the code review tasks object of this thesis.

Chapter 7 concludes this thesis by summarizing our work, highlighting its limitation and indicating future research directions based on the results we achieved.

On top of what featured in the above-described chapters, Appendix A and Appendix B present additional research done during the PhD that does not fit the topic of the thesis. In particular, Appendix A presents the research done in the context of game testing and published in the following work:

> **Using Reinforcement Learning for Load Testing of Video Games**
> Rosalia Tufano, Simone Scalabrino, Luca Pascarella, Emad Aghajani, Rocco Oliveto, Gabriele Bavota. In *Proceedings of the 44th International Conference on Software Engineering (ICSE 2022)*, pp. 2303–2314.

Appendix B, instead, presents an approach to automatically replace custom implementations with open source APIs, as described in:

> **Don't Reinvent the Wheel: Towards Automatic Replacement of Custom Implementations with APIs**
> Rosalia Tufano, Emad Aghajani, Gabriele Bavota. In *Proceedings of the 38th International Conference on Software Maintenance and Evolution (ICSME 2022)*, pp. 394-398.

2

State-of-the-art

Given the focus of this thesis on the automation of code review activities, in this chapter we present a systematic review of the literature proposing techniques and tools for the automation of code review tasks. We present the methodology we adopted to identify relevant studies and the list of selected works. Finally we discuss them, highlighting the papers that have been built on the basis of our work.

2.1 Relevant Study Identification

We describe our research method to identify the relevant studies following the guidelines by Kitchenham and Charters [KC07] for systematic literature reviews (SLR).

2.1.1 Search Strategy

We queried six digital libraries to search for primary studies: ACM Digital Library [ACM], Elsevier ScienceDirect [Els], IEEE Xplore Digital Library [IEE], Scopus [Sco], Springer Link Online Library [Spr], and Wiley Online Library [Wil]. We did not query Google Scholar due to the limitations documented by Halevi *et al.* [HMBI17a] (*e.g.,* lack of quality control, missing support for data download).

We start by performing a trial-and-error procedure to define the query needed to identify works related to the automation of code review tasks. It became soon clear that searching in the paper titles for keywords such as "*automating*", "*recommending*", etc. was not an option, even considering all their possible variations (*e.g., automating*, *automated*, *automate*). Indeed, this would led to the lost of several relevant studies (*e.g.,* "*Code Review Knowledge Perception: Fusing Multi-Features for Salient-Class Location*" [HJC+20], "*CoRA: Decomposing and Describing Tangled Code Changes for Reviewer*" [WLZX19]). For this reason, we opted for a more conservative query which targets the identification of all code review-related studies, even those do not presenting automated solutions:

Title CONTAINS
"*review*" OR ("*code*" AND "*edit*") AND
Publication venue CONTAINS
("*software*" OR "*program*" OR "*code*")

The query searches for the terms "*review*" or "*code edit*" in the article title. While only searching in the title might be restrictive, we want to identify automated solutions which have been explicitly proposed for code review (*e.g.,* we are not interested in articles presenting generic static analysis tools that might be applied in code review to spot quality issues). Also, we only searched for articles published in venues containing at least one of three keywords: "*software*", "*program*", and "*code*". Such a filter is based on our knowledge of software engineering publication venues. We acknowledge that there might be relevant articles published in related fields (*e.g.,* artificial intelligence) that our query would exclude. However, as explained later, we adopt a snowballing process to partially address this issue.

Among the queried search engines Elsevier, Scopus, Springer, and Wiley allow to specify a *discipline* of interest, which is useful to minimize the retrieved false positive instances. For these libraries, we selected "Computer Science" as discipline. Springer also allows to specify sub-disciplines, for which we selected "Software Engineering/Programming". The query has been run on 23 September 2022 on all digital libraries.

Table 2.1. Articles returned by the queried digital libraries

Source	**Returned Articles**
ACM Digital Library	885
Elsevier ScienceDirect	2,604
IEEE Xplore Digital Library	1,000
Scopus	2,041
Springer Link Online Library	1441
Wiley Online Library	64
Total (including duplicates)	8,035
Total (excluding duplicates)	**7,729**

Table 2.1 reports the articles returned by each digital library. Once removed duplicates (*i.e.,* the same article has been returned by multiple libraries), we collected 7,729 candidate primary studies which have been manually inspected as described in the following.

2.1.2 Study Selection

Given the high number of articles returned by the formulated query, we started with an automated check aimed at excluding clear false positives. First, despite the filter on venues we set in the digital libraries, we noticed that some of the returned results concerned invalid publication venues (*i.e.,* venues not featuring in their name any of the three keywords "*software*", "*program*", and "*code*"). Thus, we implemented a simple script excluding those cases (-3,784).

Other two filters were implemented. First, given our query, and in particular the retrieval of articles containing "*review*" in their title, we retrieved several SLRs. Among those we were only interested in the ones focusing on code review, since they represent an important source of references for the snowballing phase. Thus, we automatically remove all articles containing in the title, besides "*review*", the term "*systematic*" and do not containing the term "*code*" (-3,483). Second, we excluded articles published as book chapters or in magazines, since those are usually not full research articles (-153).

At the end of this process, 309 candidate primary study were left. On top of those, we added one more relevant paper we were aware of: "*AUGER: Automatically Generating Review Comments with Pre-training Models*" [LYJ+22]. This work was not found by the search because the title does not contain the combined words "code review" or "code change" but does contain "review comment". We also re-inspected all sources to make sure that searching for works with "review" and "comment" in the title did not produce any other relevant results we missed.

Table 2.2. Inclusion and exclusion criteria

	Inclusion Criteria
IC1	The article must be peer-reviewed, published at conferences, workshops, or journals. In the snowballing phase later described, we ignore all referenced preprints (*e.g.,* those published on arXiv.org).
IC2	The PDF of the article must be available online. We searched for it on the online libraries featuring and, if needed, on Google.
IC3	The article must present technique(s) to automate a code review task. It is not enough to present a generic technique that, accordingly to the reader, *might* be useful in the context of code review: The authors must explicitly state that the technique has been thought to support code review.
	Exclusion Criteria
EC1	The article is not written in English.
EC2	The article has been published in a conference/workshop and later on extended to a journal. We only keep the journal article to avoid redundancy.
EC3	The article is not a full research publication (*e.g.,* doctoral symposium articles, posters, ERA track). We exclude all articles having less than six pages with the goal of removing articles that may not have been subject to the same peer-review process typical of full research articles.
EC4	The article replicates a previously published technique for code review automation which has been already included in the SLR.
EC5	The article is a secondary study. In this case, we keep it only as a source of references for the snowballing phase.
EC6	The article has not been published in an international venue, but in a national one (*e.g., Brazilian Symposium on Programming Languages*).
EC7	The article is one of our works about code review automation ([TPT+21, TMM+22])

The set of 310 candidate primary studies has then been manually inspected. Inclusion and exclusion criteria are listed in Table 2.2. This part of the manual analysis was mainly focused on the inspection of the title and abstract of the article. We agreed to be conservative and include the article in case of doubts, given the planned subsequent reading of the whole article as described in the following.

Conflicts (*i.e.,* cases in which one researcher considered the article as relevant and one not) arisen in 17 cases (5%) and have been solved through an open discussion. This filtering process left 107 candidate studies which have been equally split among two researchers. Both researchers downloaded the corresponding article and re-inspected it keeping the inclusion and exclusion criteria in mind (Table 2.2) and then either confirming the article as relevant for the SLR or discarding it. All those discarded have been double-checked by the other researcher to ensure no relevant studies were mistakenly excluded.

This further check confirmed 59 articles as relevant primary studies. Those, together with ten articles tagged as "relevant secondary study", have been subject of a backward snowballing process.

Backward Snowballing. The 69 articles were equally split among the two involved researchers, with each of them in charge of reading the reference list and identify possible relevant papers. At this step, we retrieved also relevant papers published in venues not containing any of the three keywords "*software*", "*program*", and "*code*" (*e.g.,* papers published in the *Conference on Artificial Intelligence — AAAI*). Also in this phase, in case of doubts, the researchers agreed to included a referenced article for a further check by the other researcher. The snowballing resulted in 30 additional primary studies, that summed up to the 59 previously collected leads to the final set of 89 primary studies. The 89 primary studies have then been inspected one last time with the goal of summarizing their contribution to the state-of-the-art on code review automation.

2.2 Relevant Studies

Table 2.3 presents the 33 types of code review tasks which have been automated in the literature. It groups the tasks into macro categories (*e.g.,* "Code Change Analysis") and provides a short description of each task with related references (*i.e.,* the works addressing its automation). We discuss in the following each macro category, with a major focus on techniques targeting the tasks automated in this thesis (*i.e., code-to-code*, *code & comment-to-code*, *code-to-comment*).

Code Change Analysis

This category groups techniques aimed at analyzing the code change submitted for review in order to extract information useful to support the reviewer in its inspection. Several authors [TK15, BBaBL15, WLZX19] targeted the splitting of tangled commits [HZ13] into smaller and cohesive changes which are supposed to be easier to review. Indeed, having smaller changes can help in achieving quick review turnarounds [BB13, SSC+18] while cohesive changes simplify the identification of proper reviewers, which are more likely to have a comprehensive expertise to review the change (given its cohesiveness and focus).

Huang *et al.* [HJC+20] propose the automated identification of the "salient-class" in a commit to review. The salient-class is the one supposed to be the main focus of the changes and which likely triggered changes to other code locations. Such a class can be used as entry point for the review process, assuming that this will simplify the code change understanding.

Table 2.3. Code review tasks for which automated solutions have been proposed

Task	Description	Reference
Code Change Analysis		
Decomposing Tangled Commit	Split a composite code change into smaller and cohesive changes	[TK15, BBaBL15, WLZX19]
Predicting Salient-Class	Identification of the "salient-class" in a commit to review, namely the class causing the other changes in the commit	[HJC+20]
Linking Similar Contributions	Link similar changes to review that share textual content and modify similar code locations	[WKIM21, ACO+20]
Code Change Classification		
Predicting Code Changes Approval/Merge	Predict the likelihood of a change of being accepted (merged)	[WLZ22, WZ22, LLG+22, SLL+19, FXLL18, IAS+22]
Identifying Impactful Code Changes	Identify impactful code changes (*e.g.,* impacting the system design)	[WGRM18, UBC+21]
Identifying Large-review-effort Code Changes	Identify code changes that will require a large reviewing effort	[WBN21]
Identifying Quickly Reviewable Changes	Rank changes to be reviewed based on their likelihood of being quickly merged or rejected -	[ZdCZ19]
Code Change Quality Check		
Predicting Code Defectiveness	Predict the defectiveness of a patch before or after being reviewed	[SEB16, SS19]
Identifying Clone Refactoring Opportunities	Detect unrefactored or partially refactored code clones	[CMKS17]
Checking Design Patterns Consistency	Check whether the implemented change violates existing design patterns	[HWZ13]
Predicting Problematic Code Lines	Predict lines in a given piece of code reviewers should pay particular attention to (*i.e.,* lines likely needing changes)	[HTT22, SS23]
Reviewing via Static Analysis	Use multiple static analysis tools to generate a code review	[Bal13]
Generating Review Comments	Generate review comments for a given piece of code	[LYJ+22, LLG+22, HTTA22]
Reviewing Code Formatting Violations	Suggest how to fix code formatting violations in a given piece of code	[MLM+19]
Assessing Review Quality		
Classifying the Usefulness of Review Comments	Classify a given code review comment as useful or not-useful for the contributor	[PTPI14, RRK17, HII+21]
Identifying Review Comments Needing Further Explanations	Identifies review comments which need further explanations to be properly understood by the contributor	[RKN22]
Assessing Review Quality through Biometrics	Evaluate the quality of code review using biometrics data, warning the reviewer if specific areas of code deserve a further check	[HDC+22, HCC+21]
Code Review Sentiment Analysis		
Identifying "Pushback" Feelings in Reviews	Identify feelings of "pushback", with the reviewer blocking a change request for interpersonal conflicts	[EMHK+20]
Classifying the Sentiment of Review Comments	Classify the sentiment of review comments as neutral, negative, or positive	[ABIR17]
Identifying Toxic Code Review Comments	Identify toxic comments in code reviews	[STDB23]
Retrieval of Similar CR/CC		
Retrieving Similar Reviews	Can be used to provide either (i) the contributor with examples of reviews similar to those they are receiving (for better understanding); or (ii) the reviewer with examples of reviews which have been written for code similar to the one they are inspecting	[GS18, GYH+20, SGF+20, RKN22]
Mining Code Improvement Patterns	Extract source code improvement patterns from existing code review history to recommend how to improve the submitted code	[UIIM19]

Revised Code Generation		
Predicting the Code Output of the Review Process	Given a code snippet submitted for review, revise it to implement changes which are likely to be required by reviewers	[TPT22a, PTTC23]
Implementing the Code Change Requested by a Reviewer	Generate a revised version of a given piece of code by implementing a specific change requested by the reviewer in a natural language comment	[HHH+22, LLG+22]
Time Management		
Predicting Pull Request/Code Review Completion Time	Predict the time needed to complete a pull requests/-code review	[MBN19, SSH+22, COOM23, CRN22]
Identifying Overdue Pull Requests	Identify overdue pull requests (i.e., pull requests taking longer than the expected resolution time)	[SSH+22]
Identifying Blocking Actors in Pull Requests	Identify who among contributor(s) and reviewer(s) is to blame for overdue pull requests	[SSH+22]
Prioritizing Review Requests	Prioritize code review requests based on factors such as age of the change, test verdicts, etc.	[SB21]
Other		
Classifying the Goal of a Review Comment	Classify a review comment as Style, Functionality, Test, Approval, Disagreeing, Questioning, Roadmap, Diversion, Convention, Response or Encouragement	[LYY+17]
Generating Review Checklist	Generate a checklist to guide the reviewer's inspection	[BC96]
Recommending Reviewers	Recommend reviewers that are best suited for the given piece of code	[Bal13, TTK+15, XLWY15, OKI16, YCLW16, YWYW16, ZKB16, XSJ+17, JYH+17, FPS18, AKB+19, LWW+19, STD19, JLZ+19, MR20, ATD+20, SGBU20, COM+21, TTDE21, PT22, LLA23, AWEA23, ZMB+23, RRC16, CLBZ20, RAM+20, Ye19]
Visualizing Code Changes	Provide visualizations of the change to review to ease code comprehension	[MYG17, FS21, BV21, FFSB23]
Configuring Static Code Analysis Tools	Leverage code review comments for recommending static code analysis tools and warning categories to be used in future	[ZMA+22]

Finally, Wang *et al.* [WKIM21] suggest the automated linking of similar contributions which may help in identifying duplicated patches and, more in general, in increasing the reviewers' awareness about changes impacting similar locations, thus promoting a better code review.

Code Change Classification

Works in this area classify the whole code change to review again with the goal of augmenting the information available to reviewers before starting the code inspection. Predicting whether the code change will be approved (merged) is the most popular code change classification task tackled in the literature [WLZ22, WZ22, LLG+22, SLL+19, FXLL18, IAS+22]. Works on this topic provide a representation of the code change as input to the approach (*e.g.,* to a deep learning model) expecting it to suggest whether the implemented change is acceptable. A variation is to also provide the technique with information about the specific change the developer was asked to implement (*e.g.,* a reviewer comment that the contributor had to address).

The outputted boolean prediction can help, for example, to prioritize the diff hunks part of a pull request, focusing on those likely to require a reviewer's comment (*i.e.,* likely to be rejected [LLG+22]).

Another line of research aims at identifying code contributions which, due to their nature, will require a large review effort. Uchôa *et al.* [UBC+21] automatically flag code changes which are likely to impact the software design, thus requiring extra care in their assessment. Wen *et al.* [WGRM18] propose BLIMP Tracer, a tool to support code review through impact analysis information, thus helping in identifying changes impacting mission-critical deliverables. Wang *et al.* [WBN21] generalize the problem to the automated identification of large-review-effort changes while, at the other side of the spectrum, Zhao *et al.* [ZdCZ19] target the identification of quickly reviewable changes, namely contributions that are easy to merge or reject. Similarly to the work classifying the contributions as likely to be accepted/reject, all these works provide code reviewers with information useful for prioritizing the changes to inspect.

Code Change Quality Check

Researchers proposed solutions to (partially) automate the quality check usually in place when reviewing a code change. Approaches addressing this task substantially vary in their goal and complexity. Some of them focus on specific code quality aspects, such as predicting whether a submitted patch is likely to introduce a bug [SEB16, SS19], identifying the presence of missed clone refactoring opportunities [CMKS17], or checking whether the implemented change violates existing design patterns [HWZ13]. Other techniques address the same problem with, however, a more general view on code quality. Balachandran *et al.* [Bal13] merge the output of several static analysis tools providing the contributor with a list of potential flaws identified in the submitted patch.

With a similar goal, Hong *et al.* [HTT22] try to predict which parts of the code under review (*i.e.,* which lines). The specific issue possibly affecting the flagged lines is not reported, making the approach useful in the context of within-patch review prioritization (*i.e.,* deciding where to allocate more reviewing effort within a patch).

Markovtsev *et al.* [MLM+19] propose instead an approach that learns the code formatting style of a given software project, identifies violations to such a style, and suggests possibly fixes as automatically generated reviewer's comments.

Several recent works built on top of the research we presented at ICSE'21 [TPT+21] (Chapter 3) and at ICSE'22 [TMM+22] (Chapter 4), proposing techniques further pushing the automation capabilities in the *code-to-comment* task we targeted (*i.e.,* the automated generation of reviewer comments). Li *et al.* [LYJ+22] and Li *et al.* [LLG+22] still use pre-trained DL models to generate, given a piece of code, comments as a reviewer would do (similarly to what we do in [TMM+22]). The former exploits a solution quite similar to the one we proposed [TMM+22], with minor differences in the dataset pre-training. Instead, Li *et al.* [LLG+22] adopt a different input representation for the model as compared to our work: while our technique takes as input a single Java method to be reviewed, their approach handles diff hunks which may span several methods and provide a more compre-

hensive contextual information to the model. Also, before commenting on a given diff hunk, they verify the quality of the code under review with a boolean classifier deciding whether the code needs to be reviewed or not (*e.g.,* a diff just adding an `import` statement may be approved without the need for review).

Hong *et al.* [HTTA22] adopted a different strategy for commenting the code submitted for review as a human would do. They exploit information retrieval to recommend reviewers' comments posted in the past for code snippets similar to the one under review. This type of approach could be particularly valid for types of changes that are frequent and for which there is a good representation in the dataset.

Assessing Review Quality

While the previously discussed works focused on supporting the review process by either providing additional information to the reviewer or by taking out work from them, works in this area aim at automatically assessing the quality of the review. Such information is again meant to be fed to the reviewer who can take proper actions to improve the review quality if needed.

Works in this area can classify review comments as useful or not-useful for the contributor [PTPI14, RRK17, HII+21]. Rahman *et al.* [RKN22] addressed a similar problem but by focusing specifically on comments requiring additional explanations to be properly understood by the contributor (thus being a subcategory of not-useful comments).

Hijazi *et al.* [HDC+22, HCC+21] looked at the code review quality measurement from an orthogonal perspective using biometrics data. By monitoring the reviewer's activities (using *e.g.,* an eye-tracking device) they can provide feedback to the reviewer about areas of the reviewed code they did not pay enough attention, thus suggesting a further check.

Code Review Sentiment Analysis

The code review process may result in critiques moved by a developer (reviewer) to one of their peers (contributor). The way in which these critiques are formalized in the reviewer's comment can play an important role in the successful outcome of the whole process. For this reason, researchers applied sentiment analysis techniques to automatically classify the sentiment of reviewers' comments [ABIR17]: Flagging comments expressing a negative sentiment can provide useful information to the reviewer, who can revise those potentially problematic comments.

With a similar goal, Egelman *et al.* [EMHK+20] focus on the identification of a specific type of reviewers' comments expressing negative feelings, namely those suggesting the will of the reviewer to block a change request for interpersonal conflicts rather than for the quality of the submitted contribution.

Retrieval of Similar Code Reviews/Code Changes

Retrieval techniques have been used to create recommender systems supporting code review from different perspectives. Given a code fragment to review, some techniques [GS18, GYH+20, SGF+20] retrieve from a dataset of past reviews those involving similar code fragments and recommend to the reviewer comments they can reuse (since used in the past to suggest improvements to similar code). Rahman *et al.* [RKN22] also proposed a similar approach, but motivated it as a mechanism to provide the contributor with additional examples of reviews similar to those they are receiving. This could help in better understanding what the reviewer meant.

Ueda *et al.* [UIIM19] focused instead on mining recurring improvement patterns from code review (*i.e.,* changes frequently suggested by reviewers). Those patterns can them be potentially applied to improve the quality of the code to review (even before the review process starts).

Relevant to this task category is also the previously discussed work by Hong *et al.* [HTTA22].

Revised Code Generation

This line of research aims at supporting code review by automatically generating the code output of the code review process. In our works [TPT+21] (Chapter 3) and [TMM+22] (Chapter 4) we proposed two variations of this task with respective automated solutions. The fist provides as input to the automated technique a code snippet submitted for review and expects the technique to revise such a code to implement changes which will likely be requested during the code review process (*code-to-code*). These techniques are meant to be used by the contributor before even starting the code review process to quickly verify whether improvements can be made to the code they write. This task has been later on tackled by Thongtanunam *et al.* [TPT22a] by overcoming one of the limitations of our first proposal [TPT+21] related to the usage of code abstraction (details in Chapter 3). This work has been published in parallel (same venue) with our second contribution [TMM+22], which targeted instead several of the limitations of our first approach including, but not limited to, the one related to code abstraction.

The second variation of this task is a *code refinement* task in which the approach is provided as input not only a code snippet submitted for review but also a specific reviewer's comment to address. In this case the goal of the approach is to automatically revise the submitted code generating a version of it addressing the comment provided as input (*code & comment-to-code*). These approaches are meant to be used during the code review process either (i) by the reviewer, to attach to their comments an example of how they envision the revised code, or (ii) by the contributor, to automatically address some of the reviewer's requests.

Huq *et al.* [HHH+22] and Li *et al.* [LLG+22] built on top of our work to present more specialized [HHH+22] and performant [LLG+22] automated solutions. In particular, Huq *et al.* [HHH+22] focused on automatically fixing functional bugs identified in the code review process, thus specializing the model on this specific subset of code changes that can be learned in the context of the *code & comment-to-code* task.

Li *et al.* [LLG+22], instead, provided a broader support to developers by training their model on code written in nine programming languages (in our works we only focus on Java). Also, they consider entire code files as input to the model, as compared to the the method-level granularity adopted in our work, thus providing more contextual information to the model.

Time Management

Evidence from the literature suggests that both open source and industrial projects can undergo hundreds of reviews per month (*e.g.,* ∼500 reviews per month in Linux [RGCS14], ∼3k in Microsoft Bing [RB13]). In such a context time management becomes essential and researchers proposed solutions to help the proper allocation of reviewers' time. Differently from previously discussed techniques which automated specific code review tasks, these approaches aim at augmenting the information available to reviewers and/or managers, thus possibly improving decisions taken during code review.

Some of the proposed solutions can be combined in a sort of pipeline to support the code review: Approaches to predict the time needed to complete a pull request [MBN19, SSH+22] unlocked the possibility to identify overdue pull requests [SSH+22], namely those taking longer than expected. The identified (problematic) pull requests can, in turn, be provided as input to techniques identifying blocking actor(s) [SSH+22], namely the person(s) responsible for the delay. This could help in triggering the blocking actor or, if possible, replace them.

Still with the goal of optimizing the review time, Saini *et al.* [SB21] presented a technique to prioritize code review requests considering factors such as the age of the change, whether the change passed or not the tests, the number of revisions that have been already done for that change, etc.

Other

The last category groups together tasks which did not fit in the previously presented categories and features heterogeneous tasks. This includes the code review task which has been mostly subject to automation attempts in the literature: the recommendation of reviewers that are best suited for a change [Bal13, TTK+15, XLWY15, OKI16, YCLW16, YWYW16, ZKB16, XSJ+17, JYH+17, FPS18, AKB+19, LWW+19, STD19, JLZ+19, MR20, ATD+20, SGBU20, COM+21, TTDE21, PT22]. These techniques, while sharing the same goal, differ for the underlying technical solution adopted and for the features used to rank the reviewers given the change. In most of cases the features include information extracted from the history of code changes to favor the recommendation of reviewers who *e.g.,* already worked in the past on the code files subject of the change or already reviewed similar patches. The recency of these activities is usually considered as well.

Another popular task in the "Other" category features approaches providing visualizations for the code change to review in order to simplify the reviewer's inspection [MYG17, FS21, BV21, FFSB23]. Note that we only included in our SLR visualization techniques specifically aimed at supporting code review.

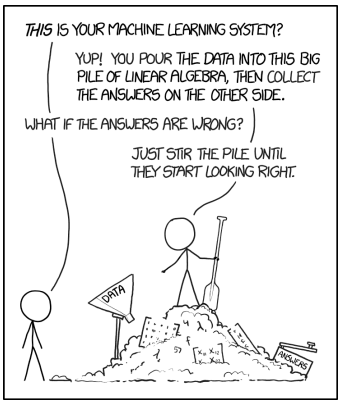
Different works focus the visualization on different types of information. Brito and Valente [BV21] propose RAID, a tool for refactoring-aware code review which visualizes the refactoring operations implemented in the change to review. Fadhel and Sekerinski [FS21] target instead visualizations aimed at improving the reviewer's awareness of the possible impact that the implemented changes can have on the system's architecture. Fregnan *et al.* [FFSB23] provide a more general-purpose graph-based visualization to support code review: Each node represents a class or a method and the links between them represents dependencies such as method calls. The goal here is to improve the navigation of the change and its comprehension. Finally, still related to visualization is the behavioral diff tool generated by the approach proposed in [MYG17]. The idea is to show the behavioral differences (in terms of test case execution) which can be observed in the system before and after the implementation of the code change to review. This can support the assessment of code change correctness made by the reviewer.

Moving to the next task, Li *et al.* [LYY+17] present an approach to automatically classify reviewers' comments into the categories reported in Table 2.3 (*e.g.,* style, functionality, etc.). Their approach is meant to provide a better understanding and monitoring of the ongoing review process. On top of that, with the proposal of data-driven techniques to automate tasks such as *generating review comments* this approach can be used to cleanup the training set of these techniques, removing for example the comments classified as "Encouragement", since irrelevant for training techniques suggesting how to improve code snippets.

Finally, Zampetti *et al.* [ZMA+22] suggest the automated analysis of review comments posted in the past to understand which static analysis tools should be used in the continuous integration pipeline of a given project and how they should be configured. In other words, they aim at understanding what are the relevant "issues" reviewers look for when inspecting a patch and which of those issues can be automatically identified by static analysis tools.

3

Using Neural Machine Translation to Automate Code Review



We present the first step we made towards automating code review by using Deep Learning (DL) models. We focus on two specific tasks. First, from the perspective of the *contributor* (*i.e.,* the developer submitting the code for review), we train a transformer model [VSP+17] to "translate" the code submitted for review into a version implementing code changes that a reviewer is likely to suggest. In other words, we learn code changes recommended by reviewers during review activities and we try to automatically implement them on the code submitted for review. This could give a fast and preliminary feedback to the contributor as soon as they submits the code. This model has been trained on 17,194 code pairs of $C_s \rightarrow C_r$ where C_s is the code submitted for review and C_r is the code implementing a specific comment provided by the reviewer. Once trained, the model can take as input a previously unseen code and recommend code changes as a reviewer would do. The used architecture is a classic encoder-decoder model with one encoder taking the submitted code as input and one decoder generating the revised source code. We name this first task *code-to-code*.

Second, from the perspective of the *reviewer*, given the code under review (C_s) we want to provide the ability to automatically generate the code C_r implementing on C_s a specific recommendation expressed in natural language (R_{nl}) by the reviewer. This would allow (i) the reviewer to automatically attach to their natural language comment a preview of how the code would look like by implementing their recommendation, and (ii) the contributor to have a better understanding of what the reviewer is recommending. For such a task, we adapt the previous architecture to use two encoders and one decoder. The two encoders take as input C_s and R_{nl}, respectively, while the decoder is still in charge of generating C_r. The model has been trained with 17,194 triplets $\langle C_s, R_{nl} \rangle \rightarrow C_r$. We name this second task *code & comment-to-code*.

Note that the two tackled problems are two sides of the same coin: In the first scenario (*i.e., code-to-code*), C_r is generated without any input provided by the reviewer, thus allowing the usage of the model even before submitting the code for review. In the second scenario (*i.e., code & comment-to-code*), C_r is generated with the specific goal of implementing a comment provided by the reviewer, thus when the code review process has already started.

We quantitatively and qualitatively evaluate the predictions provided by the two approaches. For the quantitative analysis, we assessed the ability of the models in modifying the code submitted for review exactly as done by developers during real code review activities. This means that we compare, for the same code submitted for review, the output of the manual code review process and of the models (both in the scenario where a natural language comment is provided or not as input). The qualitative analysis focuses instead on characterizing successful and unsuccessful predictions made by the two models, to better understand their limitations. The achieved results indicate that, for the *code-to-code* task (1-encoder model), the model can correctly recommend a change as a reviewer would do in 3% to 16% of cases, depending on the number of candidate recommendations it is allowed to generate. When also having available a reviewer comment in natural language (*code & comment-to-code*, 2-encoder model), the performances of the approach are boosted, with the generated code that correctly implements the reviewer's comment in 12% to 31% of cases.

The content of this chapter has been presented in the following paper:

Towards Automating Code Review Activities

Rosalia Tufano, Luca Pascarella, Michele Tufano, Denys Poshyvanyk, Gabriele Bavota. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE 2021)*, pp. 163-174.

3.1 Approach

Fig. 3.1 shows the basic steps of our approach. In a nutshell, we start by mining code reviews from Java projects hosted on GitHub [git] and/or using Gerrit [ger] as code review platform (Step 1 in Fig. 3.1). Given a code submitted by a contributor for review, we parse it to extract the list of methods it contains. Indeed, in this first work on automating code reviews, we decided to focus on small and well-defined code units represented by methods. We identify all submitted methods m_s. Then, we collect reviewer's comments made on each m_s by exploiting information available in both GitHub and Gerrit linking a reviewer comment to a specific source code line. We refer to each of those comments as r_{nl} (*i.e.,* a natural language recommendation made by a reviewer). In such a phase, a set of filters is applied to automatically discard comments unlikely to recommend and results in code changes (*e.g.,* "thank you", "well done", etc.) (2). If the contributor decides to address (some of) the received r_{nl}, this will result in a revised version of m_s addressing the received comments. We refer to such a version as m_r. Both m_s and m_r are abstracted to reduce the vocabulary size and make them more suitable for DL [TPW+19, TWB+19, WTM+20] (3).

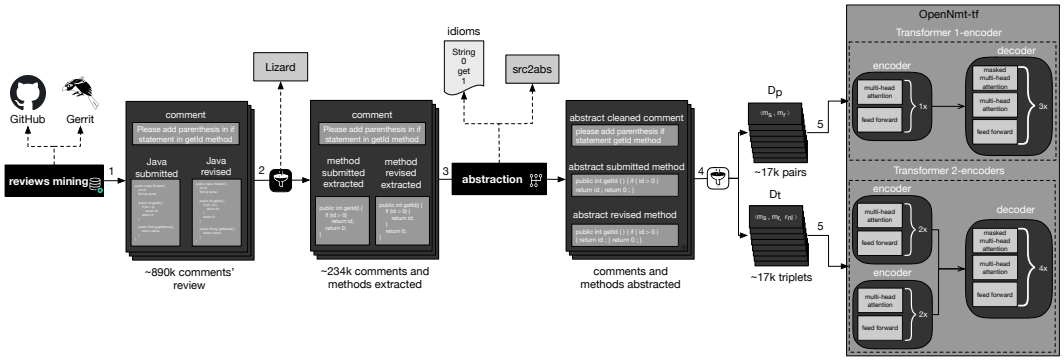


Figure 3.1. Approach overview.

To increase the likelihood that m_r actually implements in m_s a specific reviewer's comment, we only consider m_s that received a single comment in a review round. Thus, if a revised version of m_s is submitted, we can conjecture that it implements a single comment received by a reviewer (4).

Such a process results in a dataset of Reviewed Commented Code Triplets (RCCTs) in the form $\langle m_s, r_{nl} \rangle \rightarrow m_r$. Such a dataset is used to train a transformer architecture using two encoders (one processing m_s and one r_{nl}) and one decoder (generating m_r). Such a model is able, given a Java method (m_s) and a reviewer comment about it (r_{nl}) to generate a revision of m_s implementing r_{nl} (*i.e.,* m_r) (5), thus to solve the *code & comment-to-code* task.

Starting from this dataset, we also generate a dataset of code pairs $m_s \rightarrow m_r$ obtained by removing r_{nl} from each of the previous dataset triplets. This dataset has been used to train a second transformers-based model having one encoder (processing m_s) and one decoder (generating m_r). Once trained, this model can take as input a previously unseen Java method (m_s) and recommend a revised version of it (m_r) that would likely result from a review round. Since no input is required from the reviewer in this model, it can be used by the contributor to double check their implementation before submitting it for review (*code-to-code* task).

In the next sections, we describe the different steps behind our approach.

3.1.1 Mining Code Review Data

We built two crawlers for mining from Gerrit and GitHub code review data. Before moving to the technical details, it is important to clarify what the goal of this mining process is. Once a code contribution (*i.e.,* changes impacting a set of existing code files or resulting in new files) is submitted for review, it can be a subject to several review rounds. Let us assume that C_s is the set of code files submitted for review, since subject to code changes. A set of reviewer comments $\{r_{nl}\}$ can be made on C_s and, if some/all of them are addressed, this will result in a revised version of the code C_{r1}. This is what we call a "review round", and can be represented by the triplet $\langle C_s\ , \{r_{nl}\} \rangle \rightarrow C_{r1}$. The completion of a review round does not imply the end of the review process. Indeed, it is possible that additional comments are made by the reviewers on C_{r1} and that those comments are addressed.

This could result, for example, in a second triplet $\langle C_{r1}\ , \{r_{nl}\}\rangle \rightarrow C_{r2}$. The goal of our mining is to collect all triplets output of the code review rounds performed in Gerrit and in GitHub.

To this aim, we developed two miner tools tailored for systematically querying Gerrit and GitHub public APIs. The double implementation is required, because despite the fact that both platforms provide a similar support for code review, the public APIs used to retrieve data differ. Gerrit does not offer an API to retrieve all the review requests for a given project, but it is possible to retrieve them for an entire Gerrit installation (*i.e.,* an installation can host several projects, such as all Android-related projects). Starting from this information, we collect all "review rounds", and finally, we reorganized the retrieved data by associating the own set of reviews to each project. Overall, we mined six Gerrit installations, for a total of 6,388 projects.

GitHub instead offers an API to collect a list of ids of all review requests per project. In this case, we mined a set of 2,566 GitHub Java repositories having at least 50 PRs obtained by querying the GitHub APIs.

The output of this process is represented, for each review round, by (i) the set of code files submitted for review, (ii) the comments received on this code files with information about the specific impacted lines (character-level information is available for Gerrit), and (iii) the revised code files submitted in response to the received comments.

3.1.2 Data Preprocessing

After having collected the data from Gerrit and GitHub, we start its preprocessing, with the goal of building the two previously mentioned datasets of triplets ($\langle m_s,\ r_{nl}\rangle \rightarrow m_r$) for the *code & comment-to-code* task and pairs ($m_s \rightarrow m_r$) for the *code-to-code* task.

Methods Extraction and Abstraction

We start by parsing the Java files involved in the review process (both the ones submitted for review and the ones implementing the code review comments) using the Lizard [liz] Python library. The goal of the parsing is to extract the methods from all the files. Indeed, as said, we experiment with the DL models at method-level granularity, as also done in previous work [TPW$^+$19, TWB$^+$19, WTM$^+$20]. After this step, for each mined review round, we have the list of Java methods submitted for review, the reviewers' comments, and the revised list of methods resubmitted by the author to address (some of) the received comments.

Then, we adopt the abstraction process described in the work by Tufano *et al.* [TPW$^+$19] to obtain a vocabulary-limited yet expressive representation of the source code. Recent work on generating assert statements using DL [WTM$^+$20] showed that the performance of sequence-to-sequence models on code is substantially better when the code is abstracted with the procedure presented in [TPW$^+$19] and implemented in the `src2abs` tool [srcb]. Triplets for which a parsing error occur during the abstraction process on the m_s or on the m_r methods are removed from the dataset. Fig. 3.2 shows an example of abstraction procedure we perform. The top part represents the raw source code.

raw source code

```
public PageProperties getProperties() {
    if (hasProperties()) {
        return properties;
     } else {
        return null;
     }
}
```

abstracted code

```
public TYPE_1 METHOD_1 ( ) { if ( METHOD_2 ( ) )
{ return properties ; } else { return null ; } }
```

Figure 3.2. Example of abstraction.

`src2abs` uses 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). 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. We construct our list of idioms by looking for the 300 most frequent identifiers and literals in the extracted methods (list available in the replication package [repc]). The bottom part of Fig. 3.2 shows the abstracted version of the source code. Note that during the abstraction code comments are removed. `src2abs` is particularly well suited for the abstraction in our context, since it implements a "pair abstraction mode", in which a pair of methods can be provided (in our case, m_s and m_r) and the same literals/identifiers in the two methods will be abstracted using the same IDs. As output of the abstract process, `src2abs` does also provide an abstraction map M linking the abstracted token to the raw token (*e.g.,* mapping *VAR*_1 to *sum*). This allows to go back to the raw source code from the abstracted one [TPW+19].

Linking and Abstracting Reviewer Comments

Each collected reviewer comment is associated with the specific set of code lines it refers to. This holds both for Gerrit and GitHub. Using this information, we can link each comment to the specific method (if any) it refers to: Given l_s and l_e the start and the end line a given comment refers to, we link it to a method m_i if both l_s and l_e fall within m_i's body, signature, or annotations (*e.g.,* `@Override`). If a comment cannot be linked to any method (*e.g.,* it refers to an `import` statement) it is discarded from our dataset, since useless for our scope.

After having linked comments to methods for each review round, we are in the situation in which we have, for each review round, a set of triplets $\langle m_s, m_r, \text{and } \{r_{nl}\}\rangle$, where m_s and m_r represent the same abstracted method before and after the review round, and $\{r_{nl}\}$ is

a set of comments m_s received in this round. At this point, we also abstract all code components mentioned in any comment in $\{r_{nl}\}$ using the abstraction map obtained during the abstraction of m_s and m_r. Thus, assuming that the comment mentions "*change the type of sum to double*" and that the variable `sum` has been abstracted to *VAR*_1, the comment is transformed into "*change the type of VAR*_1 *to double*". On top of this, any camel case identifier that is not matched in the abstraction map but that it is present in the comment, is replaced by the special token _CODE_. Such a process ensures consistency in (i) the representation of the code and the comment that will be provided as input to the 2-encoder model, and (ii) the representation of similar comments talking about different _CODE_ elements.

Filtering Out Noisy Comments

Through a manual inspection of the code review data we collected, we noticed that a non-negligible percentage of code comments we were collecting, while linked to source code lines, were unlikely to result in code changes and, thus, irrelevant for our study. For example, if two reviewers commented on the same method, one saying "*looks good to me*" and the other one asking for a change "*please make this method static*", it is clear that any revised version of the method submitted afterwards by the contributor would be the result of implementing the second comment rather than the first one. With the goal of minimizing the amount of noisy comments (*i.e.,* comments unlikely to result in code changes) provided to our model, we devised an approach to automatically classify a comment as *likely to lead to code changes* (from now on simply *relevant*) or *unlikely to lead to code changes* (*irrelevant*).

We started by creating a dataset of comments manually labeled as *relevant* or *irrelevant*. To this aim, we randomly selected from our dataset a set of 1,875 comments and related methods m_s. These comments come from 500 reviews performed on Gerrit and 500 performed on GitHub. On our dataset (that we will detail later), such a sample guarantees a significance interval (margin of error) of ±3% with a confidence level of 99% [Ros11a]. The comments have then been loaded in a web-app we developed to support the manual analysis process, that was performed by three researchers. The web-app assigned each comment to two evaluators and, in case of conflict (*i.e.,* one evaluator said that the comment was relevant and one that was irrelevant) the comment was assigned to a third evaluator, that solved the conflict through majority voting.

Conflicts arose for 21% of the analyzed comments. Examples of comments labeled as irrelevant include simple and obvious cases such as "Thanks!" and "Nice", but also more tricky instances difficult to automatically identify (*e.g.,* "At least here it is clear that the equals method of the implementors of TreeNode is important").

The final labeled dataset consists of 1,875 comments, of which 1,676 have been labeled as *relevant* and 199 as *irrelevant*. We tried to use a simple Machine Learning (ML)-based approach to automatically classify a given comment as relevant or not. We experimented with many different variants of ML-based techniques for this task. As predictor variables (*i.e.,* features) of each comment, we considered n-grams extracted from them, with $n \in \{1, 2, 3\}$. Thus, we consider single words as well as short sequences of words (2-grams and 3-grams) in the comment.

Before extracting the *n*-grams, the comment text is preprocessed to remove punctuation symbols and put all text to lower case. In addition to this, only when extracting 1-grams, English stopwords [eng] are removed and the Porter stemmer [Por80] is applied to reduce all words to their root. These two steps are not performed in the case of 2- and 3-grams, since they could break the "meaning" of the extracted n-gram (*e.g.,* from a comment "*if condition should be inverted*" we extract the 2-gram "if condition"; by removing stopwords, the if would be removed, breaking the 2-gram). Finally, in all comments we abstract the mentioned source code components as previously explained.

After having extracted the features, we trained the *Weka* [wek] implementation of three different models, *i.e.,* the Random Forest, J48, and Bayesian network [Bre01] to classify our comments. We performed a 10-fold cross validation to assess the performance of the models. Since our dataset is substantially unbalanced (89% of the comments are relevant), we re-balanced our training sets in each of the 10-fold iterations using SMOTE [CBHK02], an oversampling method which creates synthetic samples from the minor class. We experimented each algorithm both with and without SMOTE. Also, considering the high number of features we extracted, we perform an *information gain* feature selection process [Mit97] aimed at removing all features that do not contribute to the information available for the prediction of the comment type. This procedure consists of computing the information gain of each predictor variable. This value ranges between 0 (*i.e.,* the predictor variable does not carry any useful information for the prediction) to 1 (maximum information). We remove all features having an information gain lower than 0.01.

We analyze the results with a specific focus on the precision of the approaches when classifying a comment as *relevant*. Indeed, what we really care about is that when the model classifies a comment as relevant, it is actually relevant and will not represent noise for the DL model. The achieved results reported the Random Forest classifier using the SMOTE filter as the best model, with a precision of 91.6% (meaning, that ∼92 out of 100 comments classified as *relevant* are actually relevant). While such a result may look good, it is worth noting that 89% of the comments in our dataset are *relevant*.

This means that a constant classifier always answering "relevant" would achieve a 89% precision. Thus, we experimented with a different and simpler approach. We split the dataset of 1,875 comments into two parts, representing 70% and 30% of the dataset. Then, one of the researchers tried to define simple keyword-based heuristics with the goal of maximizing the precision in classifying relevant comments on the 70% subset. Through a trial-and-error process they defined a set of heuristics that we provide in the replication package [repc]. In short, these heuristics aim at removing: (i) useless 1-word comments (*e.g.,* "nice"), (ii) requests to change formatting with no impact on code (*e.g.,* "please fix indentation"), (iii) thank you/approval messages (*e.g.,* "looks good to me"), (iv) requests to add test code, that will not result in changes to the code under review (*e.g.,* "please add tests for this method"), (v) requests for clarification (*e.g.,* "please explain"), (vi) references to a previous comment that cannot be identified (*e.g.,* "same as before"), and (vii) requests to add comments, that we ignore in our study (*e.g.,* "add to Javadoc"). Once there was no more room for improvement on the 70% subset, the set of defined heuristics has been tested on the 30% dataset, achieving precision of 93.4% in classifying relevant comments.

On the same 30% test set, the running of a Random Forest trained on the 70% dataset achieved precision of 92.1%. Given these results, we decided to use the set of defined heuristics as one of the filtering steps in our approach when preparing the dataset for training our models.

Basically, these heuristics remove from the triplets $\langle m_s, m_r, \{r_{nl}\}\rangle$ comments in $\{r_{nl}\}$ that are unlikely to have triggered the code changes that transformed m_s in m_r.

3.1.3 Automating Code Review

Dataset Preparation

Starting from the collected triplets, our goal is to build two datasets for the training/test of 1- and 2-encoder model, with the aim of solving the *code-to-code* and *code & comment-to-code* tasks respectively. First, we removed from all triplets the comments classified as noisy. Then, we built the dataset for the 2-decoder model since the other one can be easily obtained from it. We apply a set of filtering steps to obtain triplets $\langle m_s, m_r, \{r_{nl}\}\rangle$ in which:

$\{r_{nl}\}$ does not contain any comment posted by the contributor. We are interested only in reviewers' comments. Thus, author's comments are removed, leaving 231,439 valid triplets.

$\{r_{nl}\}$ does not contain any comment linked to lines in the related method representing code comments. Such r_{nl} are removed from each $\{r_{nl}\}$ before the abstraction process since, as previously explained, the abstraction removes comments.

m_s and m_r, after the abstraction, must be different. If m_s and m_r are not different, we can remove the triplet, since this means that no code change has been implemented as result of the reviewer's comments. Thus, there is nothing to learn for our models. Such a scenario can happen in the case in which the change is applied to code indentation, code comments, etc.

m_s and m_r have a reasonable length that can be handled through NMT models. The variability in sentences length can affect training and performance of NMT models even when techniques such as bucketing and padding are employed. Thus, we exclude all triplets having m_s or m_r longer than 100 tokens after abstraction. Such a filtering step has been performed in previous work [TPW+19, TWB+19, WTM+20], and it is responsible for the removal of 148,539 triplets from our dataset.

m_r does not introduce identifiers or literals that were not present in m_s. If m_r introduces, for example, a new variable `VAR_3` that was not present in m_s, in a real usage scenario it would not be possible for the model to generate the concrete raw source code for m_r, since it could not guess what the actual value for `VAR_3` should be. Thus, the model would be useless to developers. Such a limitation is due to the abstraction process that, however, has the advantage of limiting the vocabulary size and of helping the model learning [WTM+20]. However, the presence of idioms allows to retain in our dataset triplets that otherwise would be discarded because of the inability to synthesize new identifiers/literals in m_r.

$\{r_{nl}\}$ is a singleton, meaning that a single comment has been provided by a reviewer on m_s. All triplets containing more than one comment in $\{r_{nl}\}$ have been removed, since in those cases we cannot know what was the comment that triggered the transformation of m_s in m_r.

The remaining triplets are thus in the form $\langle m_s, m_r, r_{nl} \rangle$. We preprocess the r_{nl} comment to remove from it stopwords [eng], and links identified through regular expressions (*e.g.,* links pointing to online examples). Then, we clean the comment from superfluous punctuation like an ending question mark. At the end, we transform all comment words that are not code IDs in lower case, *e.g.,* the comment "*Could we use String instead of Text?*" is transformed into "*String instead TYPE_1*". Finally, we remove duplicates from the dataset.

After this process, the remaining 17,194 triplets represent what we call the D_t dataset, used for training/evaluating the 2-encoder model (*code & comment-to-code* task). By removing from each triplet the r_{nl} comment, we obtain the D_p dataset, that is instead used to train and evaluate the 1-encoder model (*code-to-code* task). Besides this difference in the two datasets, the code m_s in D_t includes two special tokens <START> and <END> which mark the part of the code interested by the reviewer's comment r_{nl}. These tokens are removed in the D_p dataset, since the 1-encoder model should be used in a scenario in which no comments have been provided by the reviewer yet. Both datasets have been split into training (80%), evaluation (10%) and test (10%) sets.

code-to-code: Recommending Changes

The 1-encoder model is meant to help the developer anticipating the changes a reviewer might suggest on the submitted code. Therefore, we want to learn how to automatically generate m_r given m_s (*i.e., code-to-code*). For this task we use a classic transformer model [VSP+17]. The transformer model consists of an encoder and decoder, which takes as input a sequence of tokens and generates another sequence as output, but it only relies on the attention-mechanism, without implying any recurrent networks. Both encoder and decoder consist of multiple layers each of which is composed of Multi-Head Attention and Feed Forward layers. In this first scenario we train a transformer model with one encoder that will take as input the sequence m_s and one decoder that will generate one or multiple suggestions for m_r.

code & comment-to-code: Implementing Changes Recommended by the Reviewer

The idea for the second scenario is to automatically implement a reviewer recommendation expressed in natural language in order to produce a practical example of what the reviewer wants. Therefore, given m_s and r_{nl} we want to automatically generate the sequence m_r (*i.e., code & comment-to-code*). Also for this task we train a transformer model, but using two encoders and one decoder. The two encoders take as input the sequences m_s and r_{nl}, respectively, while the decoder generates one or multiple suggestions for m_r. To implement both models we used the Python library OpenNmt-tf [KKD+18, opea].

Hyperparameter Search

For both models, in order to find the best configurations, we performed hyper-parameter search by adopting a Bayesian Optimization strategy [SLA12, HHL11].

Table 3.1. Hyperparameters and the best configuration

Hyperparameter	Possible values	1-encoder	2-encoder
Embedding size	[128, 256, 512, 1024, 2048]	256	512
Encoder layers	[1, 2, 3, 4]	1	2
Decoder layers	[1, 2, 3, 4]	2	4
Number of units	[128, 256, 512, 1024, 2048]	256	512
Ffn dimension	[128, 256, 512, 1024, 2048]	256	512
Number of heads	[2, 4, 8]	8	4
Learning rate	(0.0, 1.0)	0.5132	0.3370
Dropout	(0.0, 0.4)	0.2798	0.1168
Attention dropout	(0.0, 0.4)	0.1873	0.1794
Ffn dropout	(0.0, 0.4)	0.2134	0.2809

We created the space of possible configurations selecting the 10 hyper-parameters reported in Table 3.1 and choosing for each one an interval of possible values by looking at the DL literature. Given the large size of the domain space, to explore it, we chose the Tree Parzen Estimator (TPE) [BBBK11, BYC13a] as optimization algorithm with the maximum number of trials equals to 40. This means that 40 different configuration of hyper-parameters have been tested for each model. Each configuration has been trained for a maximum of 50k steps using the number of perfect predictions on the evaluation set as optimization metric. This means that the best configuration output of this process is the one for which the model is able to generate the highest number of m_r strings that are identical to the ones written by developers. To support this process, we used the Hyperopt Python library [BYC+13b, hyp].

Generating Multiple Solutions via Beam Search

Once the best configuration of each model has been selected, we evaluate it on the unseen samples of the test set. With the idea that the outputs generated by the models must be suggestions for developers/reviewers, we adopt a Beam Search decoding strategy [BCB15, BBV13, Gra12, RVY14] to generate multiple hypotheses for a given input. An output sequence is generated by adding the most likely token given the previous ones step by step. Beam search, instead of considering only the sequence of tokens with the best probability, considers the top-k more probable hypotheses, where k is known as the beam size. Thus, beam search builds k sequences simultaneously. At each timestep, it explores the space of possible hypotheses, consisting of the sequences obtainable by adding a single token to the previous k partial sequences. The process ends when the k sequences are completed. We experiment with beam sizes $k = 1, 3, 5, 10$.

3.2 Study Design

The *goal* of this study is to empirically assess whether NMT can be used to partially automate code review activities, in particular the two defined *code-to-code* and *code & comment-to-code* tasks. The *context* consists of the D_p and D_t datasets (Section 3.1).

The study aims at tackling the following research questions:

- RQ_1: *To what extent is NMT a viable approach to automatically recommend to developers code changes as reviewers would do?* This RQ focuses on the "contributor perspective" (*i.e., code-to-code* task). We evaluate the ability of an NMT model to automatically suggest code changes for a submitted code contribution as reviewers would do. We do not focus on generating the natural language comment explaining the code changes that a reviewer would require, but on providing to the developer submitting the code C_s a revised version of it (C_r) that implements changes that will be likely required in the review process. We employ the D_p dataset in the context of RQ_1.

- RQ_2: *To what extent is NMT a viable approach to automatically implement changes recommended by reviewers?* The second RQ focuses on the previously described "reviewer perspective" (*i.e., code & comment-to-code* task) and assesses the ability of the NMT model to automatically implement in a submitted code C_s a recommendation provided by a reviewer and expressed in natural language (R_{nl}), obtaining the revised code C_r. We employ the D_t dataset in the context of RQ_2.

3.2.1 Data Collection and Analysis

To answer RQ_1 we run the best configuration of the 1-encoder model obtained after hyperparameter tuning (Section 3.1.3) on the test set of the D_p dataset, and we perform an inference of the model using beam search [RVY14]. Given the code predicted by the NMT model, we consider a prediction as correct if it is identical to the code manually written by a developer after a review round (we refer to these cases as "perfect predictions"). Since we experiment with different beam sizes, we check whether a perfect prediction exists within the k generated solutions. We report the raw counts and percentages associated with perfect predictions for each beam size.

Besides reporting the perfect predictions, we also compute the BLEU-4 score [PRWZ02] of all predictions. The BLEU score is a metric used for assessing the quality of text automatically generated in the context of a NMT task [PRWZ02]. It takes values between 0% and 100%, where 100% indicates a perfect prediction, meaning that the predicted text is identical to the reference one. We use the BLEU-4 variant, computed by considering the 4-grams in the generated text and previously used in other software engineering papers (*e.g.,* [WTM+20, TWB+19]).

Also, to assess the effort needed by developers to convert a prediction generated by the model into the reference (correct) code, we compute the Levenshtein distance [Lev66] at token-level. This is the minimum number of token edits (insertions, deletions or substitutions) needed to convert the predicted code into the reference one.

Since such a measure is not normalized, it is difficult to interpret. For this reason, we normalize such a value by dividing it by the number of tokens in the longest sequence among the predicted and the reference code.

Finally, we complement our quantitative data with a qualitative analysis aimed at reporting (i) examples of perfect predictions, categorized based on the type of code change that the model automatically implemented; and (ii) non-perfect predictions, to understand whether they still can be valuable for developers. Concerning the first point, two researchers manually analyzed all 271 perfect predictions independently, and categorized them by assigning to each prediction a label describing the change automatically injected by the model. Conflicts, that arose in 11% of cases, have been solved through an open discussion. We present the obtained taxonomy as an output of this analysis. As for the second point, we use the BLEU score ranges 0-24, 25-49, 50-74 and 75-99 to split the imperfect predictions. Then, we randomly selected 25 instances from each set and two researchers manually evaluated them to determine if the recommended code change is still meaningful while being different to the reference one. Also in this case, conflicts (*i.e.,* cases in which the two researchers consistently disagreed) that arose in 9% of cases were solved through open discussion.

To answer RQ_2 we run the exact same analysis described for RQ_1, but by using the D_t dataset. The main differences are related to the performed qualitative analysis. When evaluating the perfect predictions, we decided to focus on the perfect predictions obtained by the 2-encoder model but not by the 1-encoder model. Indeed, those are most likely the cases in which the comment provided as input played a role in the prediction. For those 300 instances, the two researchers labeled the reviewers' comments to assign a label expressing the type of code change required by the reviewer. In other words, while in RQ_1 the goal was on categorizing the type of code change implemented by the model, here the focus is on the type of change requested by the reviewer in the comment. The goal is to identify categories of code comments that help the model in correctly implementing the required code change. In this case, conflicts arose for 8% of cases. The second difference, still related to the qualitative analysis, is represented by analyzed failure cases: Here the goal was to check whether the change implemented by the model, while different from the one manually implemented by the developer, was still a meaningful implementation of the change requested by the reviewer (conflicts in 2% of the analyzed instances).

3.3 Results Discussion

Table 3.2 reports the results we achieved with 1-encoder model for the *code-to-code* task (top part of Table 3.2) and with the 2-encoder model for the *code & comment-to-code* task (bottom part). It is important to remember that the two models have been experimented exactly on the same code review instances but that the 1-encoder model has been trained/tested on the D_p dataset, featuring pairs $m_s \rightarrow m_r$, while the 2-encoder model deals with the D_t dataset, composed by triplets $\langle m_s, r_{nl} \rangle \rightarrow c_r$. In other words, when generating m_r, the 2-encoder model can take advantage of the comment provided by the reviewer (r_{nl}) and asking the specific change transforming m_s into m_r, while this is not the case for the 1-encoder model.

Table 3.2. Quantitative results: Perfect predictions, BLEU-4, and Levenshtein distance achieved by the models

Beam Size	Perfect Predictions		BLEU-4			Levenshtein distance		
	#	%	mean	median	st. dev.	mean	median	st. dev.
1-encoder (*code-to-code*)								
1	50	2.91%	0.7706	0.8315	0.1929	0.2383	0.2000	0.1670
3	156	9.07%	0.8468	0.8860	0.1419	0.1726	0.1454	0.1427
5	200	11.63%	0.8644	0.8980	0.1317	0.1554	0.1271	0.1348
10	271	15.76%	0.8855	0.9145	0.1166	0.1355	0.1092	0.1247
2-encoder (*code & comment-to-code*)								
1	209	12.16%	0.8164	0.8725	0.1863	0.1849	0.1422	0.1734
3	357	20.77%	0.8762	0.9244	0.1484	0.1321	0.0838	0.1468
5	422	24.55%	0.8921	0.9376	0.1351	0.1173	0.0696	0.1366
10	528	30.72%	0.9142	0.9543	0.1169	0.0953	0.0519	0.1204

The first thing that catches the eye from the analysis of Table 3.2 are the better performance ensured by the 2-encoder model. The gap, at any level of beam size, is substantial. When only one prediction is generated (*i.e.,* $k = 1$) the 1-encoder model can generate the correct code in 50 cases (2.91% of the test set) against the 209 (12.16%) ensured by the 2-encoder model. This is a 4× improvement. The trend is confirmed for all k values, with the difference, however, becoming less strong with the increase of k. Indeed, when 10 candidate predictions are performed, 271 perfect predictions (15.76%) are generated by the 1-encoder model against the 528 (30.72%) of the 2-encoder model. While the gap in performance is still notable (+94.83% perfect predictions for the 2-enconder model), it is less marked as compared to the lowest beam size.

The BLEU-4 scores and the normalized Levenshtein distance confirm the observed trend, with the code generated by the 2-encoder model being closer to the reference code (*i.e.,* the one manually written by the developers). One observation that can be made for the 2-encoder model is that, when generating three possible previews for the code change recommended by the reviewer ($k = 3$), there is one of them requiring, on average, to only change ∼13% of the code tokens to obtain the reference code (median = 9%).

As a next step, we qualitatively analyze (i) all 271 perfect predictions obtained by the 1-encoder model with $k = 10$, and (ii) all 300 perfect predictions obtained by the 2-encoder model with $k = 10$ and for which the 1-encoder model failed to generate the correct prediction.

Table 3.3 reports a classification of the code changes performed in the 1-encoder model perfect predictions. One perfect prediction can contribute to multiple categories, since several categories of changes may be performed in a single prediction. We classified each change into two macro categories, namely *Refactoring* and *Behavioral changes*. The former groups code transformations that we judged as unlikely of resulting in behavioral changes, while the latter should impact the code behavior.

Table 3.3. Changes in the 1-encoder's perfect predictions

Refactoring (93)		
Method Visibility		**48**
	Modifies modifier	20
	Adds modifier	17
	Removes modifier	11
Readability		**42**
	Adds/Removes curly brackets	8
	Adds/Removes "this" keyword	6
	Removes unneeded variable declaration	5
	Merges two code statements	4
	Removes logging information	4
	Simplifies return statement	4
	Removes parenthesis from return statement	3
	Removes unneeded ;	2
	Removes unneeded variable cast	2
	Replaces else-if with if	1
	Replaces if-else with inline if	1
	Removes unneeded object instance	1
	Removes unneeded return statement	1
Type		**3**
	Modifies variable type	3
Behavioral changes (197)		
Code Removal		**124**
	If statement	32
	Method Invocation	31
	Return Statement	24
	Variable	21
	Deletes Method Body	15
	For Loop	1
Method Invocation		**31**
	Modifies parameters in method call	20
	Modifies method invocation	10
	Replaces method call	1
Exception Handling		**26**
	Removes thrown exception	13
	Removes try- catch	8
	Removes try- finally	4
	Moves variable assignment to finally block	1
Inheritance		**8**
	Removes invocation to parent's constructor	3
	Removes Override annotation	3
	Adds call to parent's constructor	1
	Adds modifier (final)	1
Concurrency		**6**
	Removes synchronized	6
Bug-fixing		**2**
	Modifies if condition	2

Within each macro category, a further categorization is performed to help understanding the types of code transformations learned by the model.

In the *refactoring* category, changes have been applied to the method visibility (*e.g.,* with the addition/removal/modification of `public`, `private`, etc. modifiers), to the type of variables (*e.g.,* change a variable declaration from `HashMap <String, String> VAR_1 = new HashMap<>();` to `Map <String, String> VAR_1 = new HashMap<>();`), and to improve the code readability. This sub-category features several interesting types of changes that the model recommended to simplify the code. For example, a developer submitted a method having as body `TYPE_4 <TYPE_2> reader = view.VAR_1(); return reader;`. The model recommended to *remove the unneeded variable declaration*, transforming the method body into `return view.VAR_1();`.

In the *behavioral changes* category, two of the implemented changes aimed at fixing bugs by modifying an `if` condition. For example, in one case the model added the negation (*i.e.,* `!` operator) to the if condition. Such a change was also recommended by the reviewer: "*Negation missing?*". Note that the reviewer's comment was not available to the 1-encoder model. Also other cases in the *behavioral changes* category may be related to bug-fixes but we did not have the confidence to classify them as such. For example, in the *modifies parameters in method call* category, a method invocation `message.substring(0, VAR_1+1)` was changed into `message.substring(0, VAR_1)`. The reviewer's comment mentioned: "*This line will return a substring of length maxLength + 1. If the substring needs to be no longer than maxLength, then replace "maxLength + 1" with just maxLength*" (VAR_1 maps to maxLength in the abstraction map). Thus, this is likely a bug fix, assuming that the expected behavior was the one described by the reviewer.

One message that can be derived from Table 3.3 is that the 1-encoder model is able to learn a variety of code transformations, most of them being relatively simple in terms of code changes, but sometimes solving functional/non-functional quality issues difficult to spot.

Concerning the analysis of the 2-encoder perfect predictions, here we focus on the cases in which the 1-encoder model was not able to identify the change to perform. The complete categorization of code changes we performed is available in the replication package [repc]. We present here (Table 3.4) the 20 novel categories of changes we found that were not learned by the 1-encoder model. While we anlayzed 300 instances of perfect predictions performed by the 2-encoder but not by the 1-encoder, only 32 of them (11 refactorings + 21 behavioral changes) fall into categories of changes that were completely missed by the 1-encoder. This suggests that the additional comments provided as input to the 2-encoder model, while able to substantially boost its performance (528 *vs* 271 perfect predictions when $k = 10$), do not allow it to learn many types of code changes missed by the 1-encoder. The manual analysis also gave us the opportunity to check the effectiveness of the heuristic we use to filter out irrelevant code comments.

We found that 22 out of the 300 inspected comments were irrelevant for the performed code changes (*i.e.,* were false positives that should have been discarded), leading to a ∼93% precision for our heuristic.

Table 3.4. Types of changes in the 2-encoder's perfect predictions not learned by the 1-encoder

Refactoring (9)		
Readability		**5**
	Simplify if condition	3
	Simplify if-else statement	1
	Remove unneeded null check	1
Type		**3**
	Remove type info from collection	3
Variable		**1**
	Add modifier	1
Behavioral changes (23)		
Return		**6**
	Modify return type	4
	Modify return value	2
Code Removal		**4**
	Code block	3
	Switch case	1
Exception Handling		**4**
	Modify thrown exceptions	2
	Modify try-catch	1
	Use try-with-resource pattern	1
Concurrency		**3**
	Remove concurrency lock	1
	Remove unnecessary sync guard	1
	Use shared variable instead of its copy	1
Inheritance		**3**
	Modify parent's constructor call	3
Bug-fixing		**2**
	Change value of boolean	2
Code Addition		**1**
	Add missing return	1

Looking at Table 3.4, we can see that several of the new types of changes are still simple code changes that, however, the model was only able to learn once the reviewer's comment recommending them was provided (*e.g.,* the reviewer recommended to "*use final*" as modifier for a variable, and the model successfully implemented the change). Others are instead more interesting, such as cases in which the model recommended to delete entire code blocks. Looking at them, in some cases the reviewers' were recommending an extract method, that was also suggested by the model. Clearly, the model limits the recommendation to the "source" part of the refactoring (*i.e.,* suggests which statements to extract, but not where to put them). Also, the 2-encoder model was able to learn more complex code changes that can be useful to a reviewer to quickly get a preview of how the code would look like with her comment implemented. For example, in one case the reviewer commented "*We use Java7, so you should use the try-with-resources feature*"; the 2-encoder model was able to provide as output the code implementing such a change.

As a final analysis, we looked at 100 non-perfect predictions for each model selected in the BLEU score ranges 0-24, 25-49, 50-74, and 75-99 (25 each) to determine if the recommended code change is still meaningful while being different from the reference code. For the 1-encoder, we found five (5%) of the non-perfect predictions to be still meaningful and semantically equivalent to the code written by developers (1 in the BLEU range 50-74 and 4 in 75-99). Six (6%) instances were instead found for the 2-encoder model (1 in 25-49, 3 in 50-74, and 1 in 75-99) as being successful cases of reviewer's comment implementation (despite being different from the change implemented by the developer). For instance, in one of these cases the reviewer asked to use for a public method the protected or default visibility. The developer replaced the `public` keyword with `protected`, while the 2-encoder model just removed the `public` keyword, thus using the default visibility. Overall, we can estimate an additional ∼5% of performance for the experimented models on top of what reported by the perfect predictions.

3.4 Conclusions

We experimented with DL techniques, defining two different transformer-based architectures, in the context of automating two code review activities: (i) recommending to the contributor code changes to implement as reviewers would do *before* submitting the code for review (*code-to-code*); and (ii) providing the reviewer with the code implementing a comment they have on a submitted code (*code & comment-to-code*).

While the achieved results are promising, this preliminary work has substantial limitations. First, the data used for the training and evaluation of the models. As described in Section 3.1.2, we applied several filters to foster the learning of the model and simplify the dataset, thus impacting its representativeness of real code review activities. For example, we discarded all C_s/C_r composed by more than 100 tokens, to speedup the model training and control the complexity of the learning. Second, we did not work on raw source code, but on an abstracted version which allowed us to limit the vocabulary size and avoid the model suffering from the *out-of-vocabulary* problem (*i.e.,* the model is not able to represent tokens that never show up in the training data).

While being beneficial for the learning phase, the abstraction does not allow the model to handle cases in which the transformation from $C_s \rightarrow C_r$ results in the introduction of new identifiers and literals that were not present in C_s. This means that the model cannot fix code quality issues requiring the introduction of new identifiers/literals.

These observations led us to our second work on the problem of automating code review activities described in the next Chapter of this thesis.

3.5 Replication Package

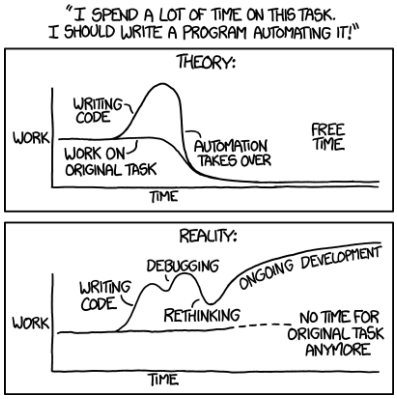
We release all code and data used in our study in a comprehensive replication package [repc]. It contains:

- the two datasets used for experimenting with the two Transformer models, as well as the raw starting data;
- all scripts used to train and test the models;
- all the obtained predictions from the trained models, as well as boxplots of the BLEU score of the predictions;
- all the utilities (*e.g.,* the list of idioms used, the list of filters we applied to the data, the logic used to remove not relevant comments);
- instructions to replicate our research.

4

Using Pre-trained Models to Boost Code Review Automation

In this Chapter we describe how we experimented with DL models for code review automation in more realistic and challenging scenarios as compared to our first work on the topic detailed in Chapter 3. The goal was to overcome the limitations of our previous work discussed in Section 3.4.



We start by training a Text-To-Text-Transfer Transformer (T5) model [RSR+20] on datasets similar to the ones described in Chapter 3. However, we adopt a tokenizer (*i.e.,* SentencePiece [KR18]) that allows us to work with raw source code, without the need for code abstraction. Also, we increase the maximum length of the considered code components from 100 "abstracted" tokens to 512 "SentencePiece" tokens (*i.e.,* ∼390 "abstracted" tokens). The absence of an abstraction mechanism and the increased upper bound for input/output length allowed us to build a substantially larger dataset as compared to the one used in Chapter 3 (168k instances *vs.* 17k) and, more importantly, to feature in such a dataset a wider variety of code transformations implemented in the code review process, including quite challenging instances such as those requiring the introduction of new identifiers and literals (accounting for 63% of the new dataset we built).

Also, we experimented with the automation of a third task related to the code review process (*code-to-comment*): Given the code submitted for review (C_s), generating a natural language comment R_{nl} requesting to the contributor code changes as a reviewer would do (*i.e.,* simulating a reviewer commenting on the submitted code).

We also compare the T5 models with the encoder-decoder models used in Chapter 3. Our results show the superior performance of T5, which represents a significant step forward in automating code review tasks. (*e.g.,* +5% and +20% of correct predictions in the *code-to-code* and *code & comment-to-code* tasks respectively, for top-1 predictions).

The content of this chapter has been presented in the following paper:

> **Using Pre-Trained Models to Boost Code Review Automation**
> Rosalia Tufano, Simone Masiero, Antonio Mastropaolo, Luca Pascarella, Denys Poshyvanyk, Gabriele Bavota. In *Proceedings of the 44th International Conference on Software Engineering (ICSE 2022)*, pp. 2291-2302

4.1 T5 to Automate Code Review

In this section we describe the DL model we adopt, the construction process of the datasets needed for its training, and the procedure used for hyperparameter search, model training, and generation of predictions.

4.1.1 Text-to-Text Transfer Transformer (T5)

The Text-to-Text Transfer Transformer, or simply T5, is not merely a model. Raffel *et al.* [RSR+20] compare "*pre-training objectives, architectures, unlabeled data sets, transfer approaches, and other factors on dozens of language understanding tasks*".

The result of this exploration is the best combination of architectures and training techniques, namely T5. T5 is based on the Transformer [VSP+17] architecture. The proposed implementation differs only in some details (regarding the normalization layer and the embedding scheme) from its original form. Raffel *et al.* proposed several versions of T5, differing from each other in their size (*e.g.,* number of layers) and, as a consequence, training complexity. In this work we adopt the *small* version of T5 consisting of: 8-headed attention, 6 layers in both the encoder and the decoder, each having a dimensionality of 512 and the output dimensionality of 2,048 ($\sim$ 60M parameters).

The model is subjected to a first training (pre-training) whose purpose is to provide it with a general knowledge useful to solve a set of related tasks. Suppose, for example, that we want to train a model able to (i) translate English to German, and (ii) summarize English text. Instead of starting by training the model for these two tasks, T5 can be pre-trained in an unsupervised manner by using the *denoising objective* (or *masked language modeling*): The model is fed with sentences having 15% of their tokens (*e.g.,* words in English sentences or code tokens in Java statements) randomly masked and it is asked to predict them. By learning how to predict the masked tokens, the model can acquire general knowledge about the language of interest. In our example, we could pre-train the model on English and German sentences.

Once pre-trained, T5 is fine-tuned on the downstream tasks in a supervised fashion. Each task is formulated in a "text-to-text" format (*i.e.,* both the input and the output of the model are represented as text). For example, for the translation task a dataset composed of pairs of English and German sentences allows to fine-tune the model. Similarly, the summarization task requires the input English text and a corresponding summary. In the next sections we explain how we pre-train and fine-tune T5 to support code review tasks.

4.1.2 Training Data

We describe the process used to build the datasets needed for the pre-training (Section 4.1.2) and fine-tuning (Section 4.1.2) of T5. Part of the fine-tuning dataset has been used for hyperparameter search (Section 4.1.3) and for testing the performance of T5 (Section 4.2).

Pre-training Dataset

Given the goal of the pre-training phase (*i.e.,* providing the model with general knowledge about the languages of the downstream tasks) we built a dataset allowing to train T5 on Java and technical English.

Indeed, besides source code, technical English is instrumental in a code review process in which reviewers post natural language comments about code.

We start from two datasets featuring instances including both source code and technical English: the official Stack Overflow dump (SOD) [sod] and CodeSearchNet (CSN) [HWG+19]. Stack Overflow is a Q&A website for programmers. The data dump we used collects all the questions and relative answers between 2006 and 2020 for a total of roughly 51M posts (where a post is a single question or answer). A post includes English text (as per the SO guidelines) and/or code snippets. Posts are usually accompanied by tags characterizing their topic (*e.g., Java*, *Android*) and can be rated with *up-/down-votes* and, for what concerns the answers, they can be marked as the "accepted answer" from the question's author.

We extracted from the SOD all the answers (i) having a *Java* tag; (ii) containing at least one `<pre><code>` HTML tag to ensure the presence of at least one code snippet in the answer; and (iii) having at least 5 up-votes and/or being the accepted answer. These filters are justified by the goal of our pre-training. Indeed, we want the model to acquire knowledge about technical English and Java: focusing on answers containing at least one code snippet increases the chances that their natural language text refers to an implementation task, similarly to what happens in code review. Also, the up-votes/accepted answer filter aims at discarding low-quality instances containing, for example, wrong code solutions. This is also the reason why we focused on high-quality answers likely to contain working solutions rather than on questions that, even if up-voted (*e.g.,* because they are relevant for many users) may contain wrong implementations. From this step we obtained 1,018,163 candidate instances from the SOD.

On each selected answer a, we performed the following cleaning steps: We remove emojis, non-latin characters, control characters, trailing spaces and multiple white spaces. Some special symbols are replaced using latin characters having the same meaning, *e.g.,* "≥" is replaced with ">=". Moreover, we replace any embedded link with a special tag "`<LINK_i>`", with i being an integer ranging from 0 to $n-1$, where n is the number of links in a. Finally, we removed all the instances having less than ten tokens or more than 512 (40,491). This left us with 977,379 valid instances.

The CSN [HWG+19] Java dataset features 1.5M unique Java methods, some of which containing their Javadoc. We filtered out all those in which a Javadoc was not available or it did not contain any letter, removing 1,034,755 of them.

Unlike the SOD, CSN can contain instances in which the "textual part" (*i.e.,* the method comment) is not in English. To partially address this issue, we exclude pairs in which no Latin characters were found. While this does not exclude all non-English comments, at least identifies and removes those written in specific languages (*e.g.,* Russian, Chinese) (15,229). We decided to accept some level of noise in the pre-training dataset (*e.g.,* comments written in French) since (i) given the size of this dataset, this little amount of noise should not substantially affect the model's performance, and (ii) the pre-training dataset is not used as test set to assess the performance of the approach. As we will explain later, a more fine-grained cleaning has been performed for the fine-tuning dataset that, instead, is used for performance evaluation. On the 519,905 remaining instances, we performed the same cleaning steps described for the SOD (*e.g.,* remove emojis). Finally, from each pair we obtain a single string concatenating the Javadoc comment and the code, retaining the ones having more than ten and less than 512 tokens (507,947 instances left).

By putting together the instances collected from the SOD and CSN we obtained the pre-training dataset consisting of 1,485,326 instances. To perform the pre-training, we randomly mask in each instance 15% of its tokens. The masked tokens are replaced with *sentinel tokens* `<extra_id_i>`, where i is an increasing number ranging from 0 up to $n-1$, where n is the number of tokens masked in a given instance. If several contiguous tokens are masked they are replaced by a single sentinel token. These "masked instances" represent the input of the model during the pre-training. The target (*i.e.,* the string the model is expected to generate) is built concatenating the sentinel tokens and the token(s) they are masking. An extra sentinel token is added to indicate the end of the string.

Fine-tuning Datasets

To create the fine-tuning dataset, this time, we mined Java open source projects from GitHub using the web application by Dabic *et al.* [DAB21]. Using the querying interface [ghs], we selected all Java projects having at least 50 pull requests (PRs), ten contributors, ten stars, and not being forks. The filters aim at (i) ensuring that enough "code review" material is contained in the projects (*i.e.,* at least 50 PRs); (ii) discarding personal/toy projects (at least ten contributors and stars); and (iii) reducing the chance of mining duplicated code. This resulted in a list of 4,901 projects. We also included the six Gerrit [ger] installations used in the previous study (see Section 3.1.1) containing code review data about 6,388 projects.

As done in Chapter 3, from both the GitHub and the Gerrit datasets we extract triplets $< m_s, r_{nl}, m_r >$, where m_s is a method submitted for the review; r_{nl} is a single reviewer's comment suggesting code changes for m_s; and m_r is the revised version of m_s implementing the reviewer's recommendation r_{nl}. Note that (i) we only looked for PRs that are accepted at the end of the code review, since we want to learn how to recommend changes that, at the end, can lead to code considered good from a reviewer's perspective; and (ii) a single PR in GitHub and Gerrit can result in several triplets for our dataset. Indeed, we mine the different review rounds in each PR. For example, a method m_s can be submitted for review, receiving a comment r_{nl} asking for changes (first round). The revised version of m_s addressing r_{nl} is then resubmitted (m_r), resulting in the second review round (possibly leading to

additional comments and revisions of the method). We stop when the code is formally accepted. Overall, we mined 382,955 valid triplets from GitHub and Gerrit using the same pipeline described in Section 3.1.2 that we summarize in the following. We target triplets in which a comment r_{nl} has been posted by a **reviewer** on a method m_s. We can identify these cases since both GitHub and Gerrit (i) provide information about the developers submitting the code and posting comments in the review process; and (ii) allow to retrieve the specific code line(s) r_{nl} refers to (*i.e.,* the code in m_s that has been highlighted by the reviewer when posting the comment).

We exclude all the comments posted by the authors of the code (*e.g.,* to reply to reviewers), since they do not represent a review of the code. Thus, the triplets in our dataset have r_{nl} being a single comment posted by a reviewer. Also, we exclude r_{nl} linked to inline comments (rather than code lines) in m_s, since we target the fixing of code-related issues. To consider a triplet as valid, r_{nl} must be the only comment posted by a reviewer on m_s in that specific review round.

In this way, we can be confident that the revised version submitted later on by the author (m_r) actually aimed at implementing r_{nl}. Also, m_r must differ from m_s (*i.e.,* a change must have been implemented in the code to address r_{nl}). From the technical point of view, the parsing of the methods from the patches submitted for review has been done using the lizard library [liz]. Note that, the removal of triplets in which r_{nl} include more than one comment has been done later in the processing pipeline (we will get back to this point). Indeed, before we had to clean comments possibly just representing noise.

As done for the pre-training dataset, we performed some cleaning steps. We replaced any link with the numbered token `<LINK_i>`, with i being an integer ranging from 0 to $n-1$, where n is the total number of links in r_{nl}, m_s and m_r. If the same link appears in different parts (*e.g.,* in r_{nl} and m_r), it is replaced with the same token. We also removed any emoji and non-ascii characters from the comments, extra spaces and control characters from both the comments and the methods, and inline comments from the methods (we are not interested in addressing issues related to internal comments).

After the cleaning process we obtained some triplets in which r_{nl} became an empty string or where m_s and m_r became equal (*e.g.,* they only differed for some spaces before the cleaning). We removed these instances (-33,005) as well as those having $r_{nl} + m_s$ or m_r longer than 512 tokens (-61,233). We considered the sum of r_{nl} and m_s in terms of length because, for one of the tasks (*i.e.,* the automated implementation of a comment posted by a reviewer), they will be concatenated to form the input for the model.

Then, we removed from our triplets non-relevant comments (-28,581), *i.e.,* comments not recommending code change suggestions (*e.g.,* "looks good to me"). In [TPT+21] we manually crafted a set of natural language patterns to spot non-relevant comments (*e.g.,* single-word comments containing words such as "thanks", "nice", etc.). We have extended this set since we noticed that in our richer dataset several non-relevant comments were left by these patterns. Such analysis has been done by manually inspecting all the triplets having r_{nl} consisting of less than six words. The updated heuristics are available in our replication package [repd]. We also excluded triplets including non-English r_{nl} comments (-4,815) through a pipeline composed by three language detector tools.

A preliminary classification has been performed using the Python libraries langdetect [lana] and pycld3 [pyc]. If both of these tools classify the comment as non-English, we relied on the Google language detection API for a final decision. Such a process was needed since we noticed that the Google API was the most accurate in detecting the language, especially when the comments also featured code constructs in them. In this scenario, the Python libraries often generated false negatives (*i.e.,* classifying an English sentence as non-English). However, we had a limited number of requests available for the Google API. Thus, we performed a pre-filtering using the Python libraries and, when they both reported the comment as being not in English, we double checked using the Google API.

After this cleaning process, we excluded all triplets featuring more than one comment in r_{nl} (-86,604). Finally, we removed all the duplicates from the fine-tuning dataset (-918). To be conservative, we identify as duplicates two triplets having the same m_s (thus, even triplets having the same m_s but different r_{nl}/m_r have been removed).

The resulting dataset features 167,799 triplets that have been used to build the three fine-tuning datasets needed for the three tasks we aim at automating. In the first task (*code-to-code*) the model takes as input m_s with the goal of automatically generating its revised version m_r, implementing code changes that may be required in the code review process. Thus, the fine-tuning dataset is represented by pairs $m_s \rightarrow m_r$.

In the second task (*code&comment-to-code*) the model takes as input both m_s and a comment r_{nl} posted by the reviewer and targets the generation of m_r, the revised version of m_s implementing the code changes recommended in r_{nl}.

The m_s code contains two special tags `<START>`, `<END>` marking the portion of the code r_{nl} refers to. The fine-tuning dataset of this second task is represented by pairs $< m_s, r_{nl} > \rightarrow m_r$.

Finally, in the third task (*code-to-comment*) the model takes as input m_s and aims at generating a natural language comment (r_{nl}) suggesting code changes as a reviewer would do. The fine-tuning dataset is represented by pairs $m_s \rightarrow r_{nl}$.

Table 4.1. Pre-training and fine-tuning datasets (# instances)

Dataset	train	evaluation	test
Pre-training			
Stack Overflow	977,379	-	-
CodeSearchNet	507,947	-	-
Fine-tuning	134,239	16,780	16,780

All three fine-tuning datasets have been split into 80% training, 10% evaluation, and 10% test. Table Table 4.1 summarizes the number of instances in the datasets: The pre-training is only used for training, while the fine-tuning datasets are exploited also for the hyperparameter tuning (*evaluation*) and for assessing the performance of the model (*test*). In Table 4.1 we only report information for a single fine-tuning dataset (rather than for the three previously described), since all three fine-tuning datasets contain the same number of instances. Indeed, they are all derived from the same set of triplets.

4.1.3 Training and Hyperparameter Search

Raffel *et al.* [RSR+20] showed the major role pre-training plays on the performance of T5 models. The importance of pre-training has also been confirmed (for other Transformer-based models) in the context of code-related tasks such as test case generation [TDS+20]. To further study this aspect, we decided to experiment with both a pre-trained and a non pre-trained model, both of which have been subject to a hyperparameter tuning process.

Since we adopted the *small* version of T5 presented by Raffel *et al.* [RSR+20], we did not experiment with variations related to its architecture (*e.g.,* changing the number of layers or the number of hidden units). Though, as also done by Mastropaolo *et al.* [MSC+21], we experimented with different learning rate configurations: (i) *Costant Learning Rate* (C-LR), in which the learning rate value is fixed during the training; (ii) *Inverse Square Root Learning Rate* (ISR-LR), in which the learning rate value decays as the inverse square root of the training step; (iii) *Slanted Triangular Learning Rate* (ST-LR) in which first the learning rate linearly increases and then it linearly decays returning to the starting value; (iv) *Polynomial Decay Learning Rate* (PD-LR), in which the learning rate polynomially decays to a fixed value in a given number of steps.

The hyperparameter tuning has been done for the fine-tuning phase only. Indeed, even though we just focus on one hyperparameter, such a process still remains quite expensive, requiring the training of eight different T5 models (*i.e.,* pre-trained and non pre-trained each with four different learning rates).

For pre-training we use the same configuration proposed by Raffel *et al.* in [RSR+20]. We pre-trainied the model on the pre-training dataset (Table 4.1) for 200k steps (∼34 epochs). Starting from the pre-trained model, we fine-tuned for 75k steps four different models, each using one of the experimented learning rates.

Since the goal of this procedure is to find the best learning rate for the three code review tasks, we fine-tuned each of these models using a mixture of the three tasks: A single model is trained to support all three tasks using the union of their training sets. This is one of the characteristics of T5, the possibility to train a single model for multiple tasks. The same approach has been used for the non pre-trained model: In this case four T5 models (one per learning rate) have been directly fine-tuned.

We assessed the performance of the eight models on the evaluation set of each task in terms of "perfect predictions", namely cases in which the generated output was identical to the target (expected) string. Table 4.2 reports the achieved results. As it can be seen, no learning rate achieves the best results in all the tasks. Nevertheless, ST-LR shows better overall performance and, for this reason, is the one we adopt in our experiments.

Given the best configuration for both the pre-trained and the non pre-trained models, we fine-tuned them for a maximum of 300k steps using an *early stop strategy*. This means that we saved a checkpoint of the model every 10k steps computing its performance in terms of "perfect predictions" on the evaluation set and stopped the training if the performance of the model did not increase for three consecutive checkpoints (to avoid overfitting).

Table 4.2. Hyperparameter tuning results

Task	Learining Rate Strategy			
	C-LR	ISR-LR	ST-LR	PD-LR
Pre-Trained				
code-to-code	2.68%	3.68%	**4.64%**	2.53%
code&comment-to-code	**10.39%**	9.23%	8.46%	9.89%
code-to-comment	0.15%	0.32%	**0.60%**	0.15%
Non Pre-Trained				
code-to-code	1.23%	3.71%	**4.16%**	1.22%
code&comment-to-code	5.05%	**6.41%**	6.24%	5.18%
code-to-comment	0.09%	0.44%	**0.49%**	0.03%

4.1.4 Generating Predictions

Once the models are trained, they can be used to generate predictions. As done in previous work, we adopt a beam search strategy [RVY14] to generate multiple predictions given a single input. For example, in the case of the *code-to-code* task, for a single m_s method provided as input multiple m_r candidates can be generated. When we ask the model to generate k predictions, it generates the k most probable sequences of tokens given the input sequence; k is known as the *beam size* and we experiment with $k = 1, 3, 5, 10$.

For each prediction generated by T5, we also exploited its `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 be 0. In our empirical study (Section 4.2) we assess the reliability of the confidence level as a proxy for the quality of the predictions.

4.2 Study Design

The *goal* of our evaluation is to empirically assess the performance of the T5 model in code review automation tasks. The *context* consists of (i) the datasets we presented in Section 4.1.2; and (ii) the dataset from [TPT+21] described in Chapter 3. From now on we refer to our previously presented approach as the *baseline*. The study aims at tackling five research questions (RQs).

RQ_1: *To what extent is T5 able to automatically recommend code changes to developers as reviewers would do?* We provide as input to T5 a Java method m_s submitted for review and assess the extent to which the model is able to provide as output a revised version of m_s (m_r) implementing code changes that will be likely requested during the code review process. The idea here is that such a model could be used *before* the code is submitted for review as an automated check for the contributor.

RQ$_2$: *To what extent is T5 able to automatically implement code changes recommended by reviewers?* Given a Java method submitted for review (m_s) and a natural language comment (r_{nl}) in which a reviewer asks to implement specific code changes in m_s, we assess the ability of T5 to automatically revise m_s to address r_{nl} (thus obtaining a revised method m_r).

The third RQ focuses on the novel code review-related task we introduce in this paper:

RQ$_3$: *To what extent is T5 able to automatically recommend changes in natural language as reviewers would do?* In this RQ T5 is provided as input with a Java method submitted for review (m_s) and it is required to generate a natural language comment (r_{nl}) requesting code changes as reviewers would do.

For RQ$_1$-RQ$_3$, we experiment with different variants of the T5 model. In particular, we assess the quality of T5 predictions for all three tasks when (i) the model is pre-trained or not; and (ii) the predictions have different confidence levels. Thanks to these analyses, we can answer our fourth RQ:

RQ$_4$: *What is the role played by the model pre-training on the performance of T5? How does the confidence of the predictions affects their quality?* As explained in Section 4.1.3, we perform an ablation study in which T5 is fine-tuned without any pre-training (*i.e.,* by starting from random weights in the neural network). This allows to assess the contribution of the pre-training to the performance of the model. As for the confidence of the predictions, we assess whether it can be used as a reliable proxy for the quality of the predictions (*i.e.,* the higher the confidence, the higher the likelihood the prediction is correct). If this is the case, such a finding would have implications for the usage of the T5 model in practice: A developer using the model could decide to receive recommendations having confidence higher than t, reducing the chances of receiving meaningless predictions.

Finally, the last RQ compares the performance of the T5 model with that of the approach we presented in Chapter 3:

RQ$_5$: *What is the performance of T5 as compared to the state-of-the-art technique?* We use the implementation and datasets from our previous work to compare the performance of the T5 model with the baseline [TPT$^+$21].

4.2.1 Data Collection and Analysis

To answer the first four research questions, we experiment with the best configuration of both the pre-trained and non pre-trained T5 model on the test set of the fine-tuning dataset reported in Table 4.1. Remember that for each of the three tasks we support (*i.e.,* the ones that map to RQ$_1$, RQ$_2$, and RQ$_3$) the 16,779 test set instances are the same triplets $< m_s, r_{nl}, m_r >$. The only difference is that: in RQ$_1$ the model has been trained (and is tested) to take as input m_s and produce m_r; in RQ$_2$ it takes as input m_s and r_{nl} and produces m_r; in RQ$_3$ it takes as input m_s and produces r_{nl}. By running the models on the test sets, we report for each of the three tasks the percentage of "perfect predictions", namely the cases in which the output of the model is the expected one. For example, in the case of RQ$_3$, this means that the model was able, given m_s as input, to generate a comment r_{nl} identical to the one manually written by the reviewer who inspected m_s.

Besides computing the perfect predictions, in RQ_3 (*i.e.,* the task in which the model is required to generate natural language text), we also compute the BLEU (Bilingual Evaluation Understudy) score of the predictions [PRWZ02]. BLEU assesses the quality of the automatically generated text. The BLEU score ranges between 0 and 1, with 1 indicating, in our case, that the natural language comment generated by the model is identical to the one manually written by the reviewer. We use the BLEU-4 variant, that computes the overlap in terms of 4-grams between the generated and the reference text.

In RQ_1 and RQ_2 (*i.e.,* in the tasks in which the model is required to generate code), we adopt instead the CodeBLEU [RGL+20], a recently proposed similarity metric inspired by the BLEU score but tailored to assess the quality of automatically generated code.

Differently from BLEU, CodeBLEU computes not only an "n-gram based similarity" but it also considers how similar the abstract syntax tree and the data-flow of the generated and the reference code are. Ren *et al.* [RGL+20], who proposed the CodeBLEU, showed that their metric better correlates with developers' perception of code similarity as compared to the BLEU metric.

Concerning RQ_4, we compare the results (*i.e.,* perfect predictions, BLEU, CodeBLEU) achieved by the T5 model with and without pre-training. We also statistically compare the two models (*i.e.,* with/without pre-training) using the McNemar's test [McN47] and Odds Ratios (ORs) on the perfect predictions they can generate. As for the confidence of the predictions, we take the best performing model (*i.e.,* the one with pre-training) and split its predictions into ten buckets based on their confidence c going from 0.0 to 1.0 at steps of 0.1 (*i.e.,* the first interval includes all predictions having a confidence c with $0 < c \leq 0.1$, the last interval has $0.9 < c \leq 1$). Then, we report for each interval the percentage of perfect predictions.

Finally, in RQ_5, we compare T5 with the baseline [TPT+21] on the two tasks automated in our previous work (*i.e.,* the ones related to our RQ_1 and RQ_2).

As metrics for the comparisons, we used the percentage of perfect predictions and the CodeBLEU of the predictions. We compared the two techniques in several scenarios. First, we used the dataset from [TPT+21] featuring 17,194 triplets $< m_s, r_{nl}, m_r >$. By performing some checks on this dataset, we noticed that a few instances (97) had comments (r_{nl}) not written in English or containing invalid unicode characters that did not allow our tokenizer to work. Thus, we excluded those instances from the training and the test sets. The training set has then been used to (i) train the baseline [TPT+21]; and (ii) fine-tune the T5 model without any pre-training. In this way, we can compare the performance of the two models on the test set when trained on exactly the same data. Important to notice is that the baseline has been trained and tested on abstracted code (as done in [TPT+21]), while T5 worked directly with the raw source code.

On top of this, we also report the performance of the pre-trained T5 model when run on the test set from [TPT+21]. This pre-trained model has been fine-tuned using the training dataset in [TPT+21]. Clearly, this analysis favors T5 since it has been trained on more data (*i.e.,* the pre-training dataset). However, it provides additional hints into the role played by the pre-training and on the effectiveness of the T5 model in general.

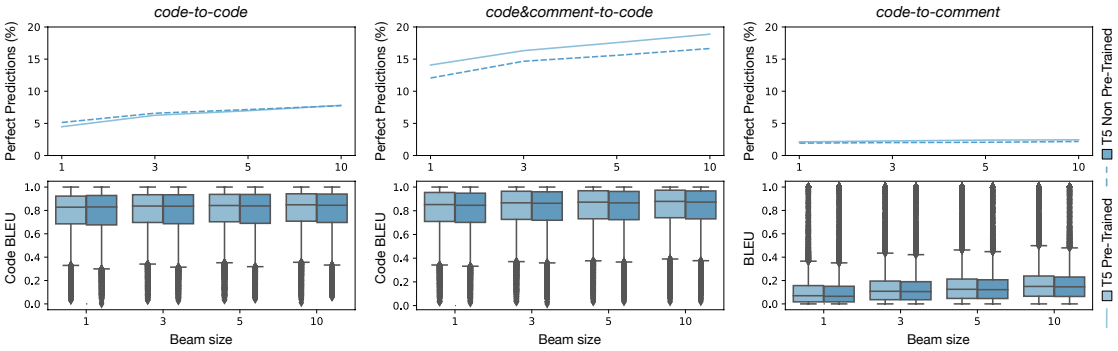


Figure 4.1. Results T5 dataset large

Besides reporting descriptive statistics, we statistically compare the two models using the McNemar's test [McN47] and Odds Ratios (ORs) on the perfect predictions they can generate. Since multiple comparisons are involved (*e.g.,* comparing the pre-trained and the non pre-trained model to the baseline), we adjust the p-values using the Holm's correction [Hol79b].

4.3 Results Discussion

We start by answering **RQ$_1$-RQ$_3$** (Section 4.3.1), presenting the performance of T5 in the three tasks we aim at automating. Then, we discuss the impact on the performance of the pre-training and the reliability of the confidence level as a proxy for the quality of the predictions (Section 4.3.2). Finally, we compare T5 with the baseline [TPT$^+$21] (Section 4.3.3).

4.3.1 RQ$_1$-RQ$_3$: Performance of T5

Fig. 4.1 reports two graphs for each task. The line chart on top shows the percentage of perfect predictions (y-axis) achieved by T5 for different beam sizes (x-axis); the continuous line represents the pre-trained version of the model, while the dashed line the non pre-trained one. The boxplots at the bottom report the CodeBLEU for the two code-generation tasks (*i.e., code-to-code* and *code & comment-to-code*) and the BLEU score for the *code-to-comment* task in which text is generated. Lighter blue represents the pre-trained model.

We start by commenting on the perfect predictions (line charts). At a first sight, the performance of the model might seem quite low. For example, in the case of *code-to-code* at $k = 1$ (*i.e.,* a single prediction is proposed by T5), both the pre-trained and the non pre-trained models achieve ∼5% of perfect predictions (751 and 863 instances correctly predicted with and without pre-training, respectively). However, such a result should be considered in the context of what was reported by the state-of-the-art technique [TPT$^+$21] that, on a much simpler test dataset, achieved for the same task and same beam size 2.91% of perfect predictions.

Figure 4.2. Examples of perfect and alternative predictions

Similar observations can be made for the *code & comment-to-code* task, where at $k = 1$ T5 can generate 14.08% (2,363 instances) and 12.06% (2,024) perfect predictions when pre-trained and not, respectively. For this task, in our previous work [TPT+21], we achieved on a simpler dataset 12.16% perfect predictions. We directly compare the two approaches in RQ_5. Interestingly, increasing the beam size from 1 to 10 does only result in marginal improvements for all tasks. The largest improvement is obtained for the *code & comment-to-code*, where we move from 14.08% ($k = 1$) to 18.88% ($k = 10$) of perfect predictions for the pre-trained model. Given the goal of our approach, we believe that the most relevant performance are those achieved at $k = 1$. Indeed, providing several recommendations to inspect to a developer might be counterproductive, especially considering that the recommendations are entire methods in the case of the two code-generation tasks.

Moving to the *code-to-comment* task, T5 struggles in formulating natural language comments identical to the ones written by reviewers. The pre-trained model, at $k = 1$, generates 356 correct comments (2.12%) against the 324 (1.93%) of the non pre-trained model. These numbers only slightly increase at $k = 10$, with a maximum of 2.44% perfect predictions achieved with pre-training.

The top part of Fig. 4.2 shows two examples of perfect predictions generated by the model for each task. A dashed line separates the two examples within each task. For the *code-to-code* task, the first code in each example represents the input of the model, while the second its output. We highlighted in bold the parts of code changed by the model and replaced irrelevant parts of the methods with [...] to save space.

In the first *code-to-code* example, T5 removes an unneeded `instanceof` check, since `FileSystemDataset` is a subclass of `Dataset`. Instead, the second example simplifies the checking for the existence of a cluster, providing a meaningful error message. This second case cannot be supported by the baseline [TPT+21], since it requires the introduction of new code tokens that were not present in the input code. Remember that, these being perfect predictions, the implemented changes are identical to those performed by developers during code review.

For the *code & comment-to-code* task, the input provided by the model includes the comment written by the reviewer and requiring a specific change to the part of code highlighted in orange. In the first example, the reviewer suggests to use a specific object to perform the `null` check and T5 correctly implements the change. The second one is interesting because, despite the reviewer highlighting `return null` as the relevant code for their comment ("*else is redundant*"), the model correctly understands that the action to take is the removal of the unneeded `else` statement.

Finally, for the *code-to-comment* task, we report the code provided as input to the model (first line) with the comment it generated as output (second line). In the first example, T5 suggests (as done by the real reviewer) to add a `null` check, also showing the code needed for its implementation. This code is not just a template, but it is suitable for the provided input code (it refers to the `supplier` object). In the second example, T5 suggests to rename an identifier, providing valid recommendations for the renaming.

Looking at the bottom of Fig. 4.1, the results in terms of CodeBLEU show a median higher than 0.80 for all beam sizes and for both code-generation tasks. However, while we report these values for completeness and for being consistent with what done in similar works [TWB+19, WTM+20, TPT+21], they say little about the quality of the predictions and they are mostly useful for future work that wants to compare with our approach (complete distributions are available in our replication package [repd]). Indeed, it is difficult to properly interpret these values for two reasons. First, there is no accepted threshold above which good performance can be claimed. Second, as also done in previous works proposing models taking as input a code snippet and providing as output the same code "revised" in some way (*e.g.,* with a fixed bug [TWB+19], with a single statement added [WTM+20], or with review-related changes implemented [TPT+21]), we computed the CodeBLEU between the predicted and the target code (two methods in our case). However, the input provided to the model is already quite similar to the target output, which means that a model taking as input a method and not implementing any change on it, is likely to obtain high values of CodeBLEU. For this reason, we mostly focus our discussion on perfect predictions. Concerning the BLEU score achieved in the *code-to-comment* task, the median ranges around 0.10 (see Fig. 4.1). Such a result is expected given the low percentage of perfect predictions achieved for this task.

Going back to the perfect predictions, the results reported in the line charts in Fig. 4.1 represent a lower bound for the performance of our approach. Indeed, we consider a prediction as "perfect" only if it is identical to the reference one. For example, in the case of the *code-to-comment* task, the natural language comment generated by T5 is classified as correct only if it is equal to the reference one, including punctuation.

However, it is possible that a natural language comment generated by T5 is different but semantically equivalent to the one written by the developer (*e.g.,* "variable *v* should be private" *vs* "change *v* visibility to private"). Similar observations hold for the two code-generation tasks (*e.g.,* a reviewer's comment could be addressed in different but semantically equivalent ways).

To have an idea on the number of valuable predictions present among those classified as "wrong" (*i.e.,* the non-perfect predictions), three researchers manually analyzed a sample of 100 "wrong" predictions for each task (300 in total). The analysis was done in two meetings in which each instance was discussed by all three researchers. The goal was to classify each instance into one of three categories: (i) "semantically equivalent" (*i.e.,* the generated code/comment is different but semantically equivalent to the reference one); (ii) "alternative solution" (*i.e.,* the generated code/comment is not semantically equivalent, but valuable); or (iii) "wrong" (*i.e.,* the generated code/comment is not meaningful for the provided input). Since we also computed the confidence for each of the predictions generated by T5, rather than randomly selecting the 300 instances to inspect, we decided to target for each task the top-100 wrong predictions generated by the model in terms of confidence. Indeed, those cases are particularly interesting, since they represent wrong predictions for which, however, the model is quite confident.

Table 4.3. Manual analysis of 100 "wrong" predictions per task

Task	**Semantically Equivalent**	**Alternative Solution**	**Wrong**
code-to-code	1	10	89
code & comment-to-code	6	56	38
code-to-comment	36	10	54

Table 4.3 shows the results of our manual analysis. For the *code-to-code* we observed that, in most cases (89%) the model actually generates wrong predictions that are not inline with the changes implemented by the developer. There are few exceptions to these cases, mostly related to small changes in which the model made a decision different from that one of the developer but still valid (*e.g.,* extracting a string into a variable and using a different name for the extracted variable). More interesting are the results for the other two tasks.

In the case of *code & comment-to-code*, we found that 62 out of the 100 "wrong" predictions we inspected were actually valid implementations of the change recommended by the reviewer. One example is presented at the bottom of Fig. 4.2 (black background), where we show the input provided to the model (*i.e.,* the code in the first line and the reviewer's comment "*Inline this variable*") and the output of the model right below. T5 successfully addressed the reviewer's comment.

However, the prediction is different from the target implementation, since the latter also includes another change that was not explicitly required in the code review. This case is representative of all 56 instances we classified as "alternative solutions" for this task and, given the goal of the *code & comment-to-code*, we believe they represent good predictions.

Finally, also for the *code-to-comment* task, we found a large number of "wrong" predictions that are actually valuable, with 36 of them even being semantically equivalent (*i.e.,* T5 formulated a comment asking the same changes required by the reviewer, but using a different wording). One example is reported at the very bottom of Fig. 4.2. While the model only received the code as input we also show the original reviewer's comment (*i.e., "Please make this one a variable as well"*) to make it easier to assess the relevance of the comment generated by T5 (*i.e., "Extract the building ..."*).

Overall, our analysis showed that the perfect predictions really represent a lower bound for the performance of T5, especially for the two tasks in which natural language comments are involved.

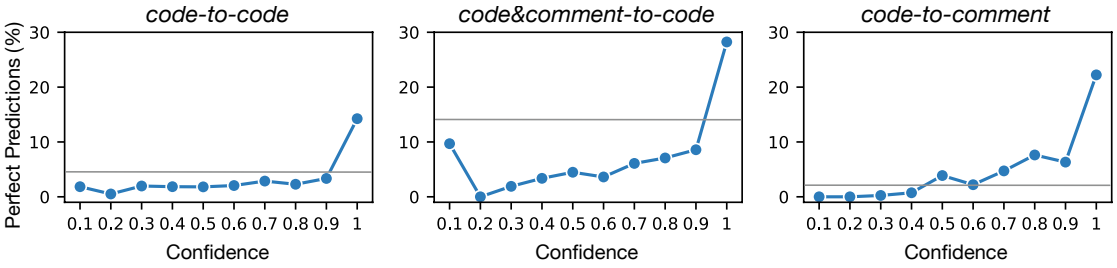


Figure 4.3. Perfect predictions by confidence of the model

4.3.2 RQ_4: Pre-training and confidence

In Fig. 4.1 we observed better performance for the pre-trained model in the *code & comment-to-code* and in the *code-to-comment* task, while the non pre-trained model performed better in the *code-to-code* task. The results of the McNemar's test on the predictions at k=1, confirm such findings: besides the significant difference confirmed for all tasks (p-value $<$ 0.01), the ORs indicate 85% and 59% higher odds of obtaining a perfect prediction using the pre-trained model in the *code & comment-to-code* (OR=1.85) and in the *code-to-comment* (OR=1.59) task, while odds are 34% lower in the *code-to-code* task (OR=0.66).

Two observations are worth to be made. First, overall, the pre-trained model seems to represent a more valuable solution. Second, the lack of improvement in the *code-to-code* task can be explained by the pre-training and fine-tuning we performed. Indeed, the *code-to-code* task only focuses on source code, with no natural language in the input nor in the output. The fine-tuning stage, focused on source code, was probably sufficient to the model to learn about the code syntax and the possible transformations to perform. The additional pre-training, also including technical English, did not benefit the model for the *code-to-code* task. The other two tasks, instead, either include natural language as input (*code & comment-to-code*) or require its generation as output (*code-to-comment*), obtaining a boost of performance from the pre-training.

Fig. 4.3 depicts the percentage of perfect predictions (y-axis) within each confidence interval (from 0.0-0.1 up to 0.9-1.0, x-axis) when using the pre-trained model and k=1. To better interpret the reported results, the gray line represents the overall performance of the

model when considering all predictions (*e.g.,* 4.48% of perfect predictions for the *code-to-code* task).

In all three tasks, we observe a clear trend, with the predictions in the highest confidence bucket (0.9-1.0) ensuring substantially better performance than the overall trend. When only considering the predictions in this bucket, the percentage of perfect predictions increases to: 14.24% for *code-to-code* (from an overall 4.48%), 28.23% for *code & comment-to-code* (overall=14.08%), and 22.23% for *code-to-comment* (overall=2.12%). Considering the complexity of the addressed tasks, the jump in performance is substantial and indicates the usability of the confidence level as a proxy for the prediction quality. Also, while the percentage of perfect predictions is quite limited, with seven out of ten predictions being wrong in the best-case scenario (28.23% for *code & comment-to-code*), it is worth considering what previously observed in our manual analysis, with "valuable" predictions which are classified as "wrong" in our quantitative analysis.

4.3.3 RQ_5: Comparison with the baseline [TPT+21]

Fig. 4.4 compares the performance achieved by the T5 model with those obtained by the baseline [TPT+21]. In the line charts the continuous lines represent the pre-trained T5, the dashed lines non pre-trained T5, and the dotted lines the baseline.

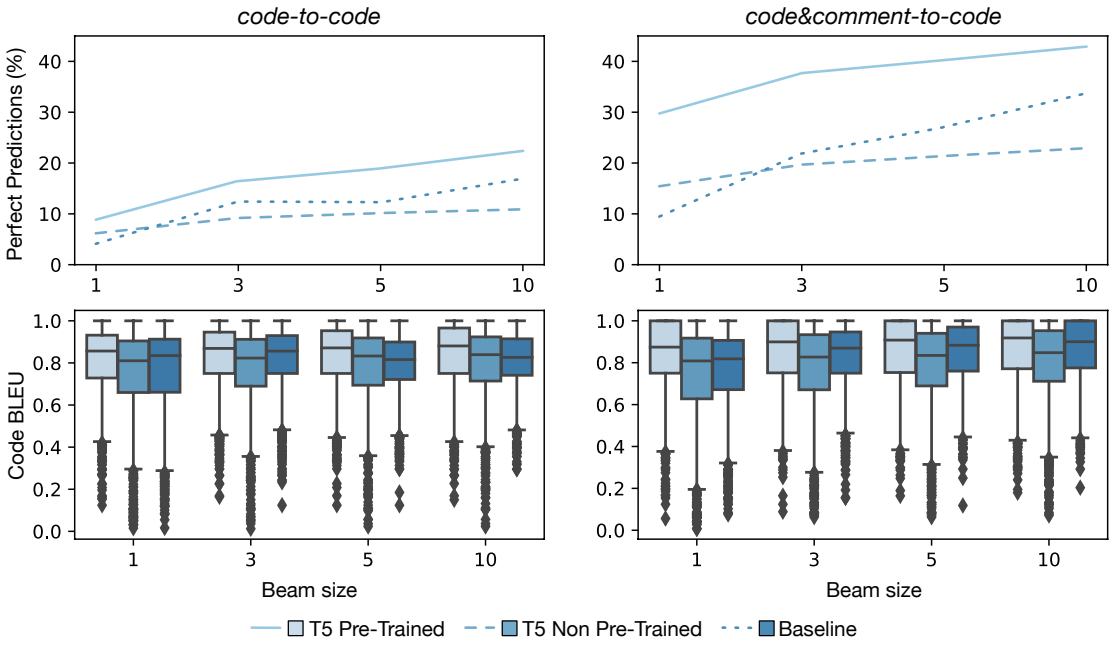


Figure 4.4. T5 *vs.* baseline [TPT+21]

Table 4.4. RQ_5: McNemar's test (adj. *p*-value and OR)

Task	Test	*p*-value	OR
code-to-code	T5 pre-trained *vs* [TPT$^+$21]	<0.01	2.90
	T5 non pre-trained *vs* [TPT$^+$21]	<0.01	1.69
	T5 pre-trained *vs* T5 non pre-trained	<0.01	2.50
code & comment-to-code	T5 pre-trained *vs* [TPT$^+$21]	<0.01	11.48
	T5 non pre-trained *vs* [TPT$^+$21]	<0.01	2.38
	T5 pre-trained *vs* T5 non pre-trained	<0.01	5.69

Two important points are worth remembering: First, the results in Fig. 4.4 have been computed on the test set used in [TPT$^+$21]. Indeed, the performance in terms of perfect predictions are substantially higher as compared to those in Fig. 4.1 (see values on the *y*-axis), due to the simpler instances featured in this dataset. Second, the baseline has been trained and tested on abstracted code (as in the original paper), while T5 worked on raw source code.

When k=1, T5 achieves substantially better performance. The results of the statistical test in Table 4.4 always show a significant difference in favor of T5 (adjusted *p*-value $<$ 0.01), with ORs ranging from 1.69 (non pre-trained T5 *vs* [TPT$^+$21] in the *code-to-code* task) to 11.48 (pre-trained T5 *vs* [TPT$^+$21] in the *code & comment-to-code* task). The pre-trained T5 in this case performs better than the non pre-trained one for both tasks. This is likely due to the limited size of the fine-tuning dataset used in this comparison. Indeed, to have a fair comparison with [TPT$^+$21], we fine-tuned T5 on the training set we used in [TPT$^+$21] and composed by ∼13.5k instances (*vs* the ∼134k we had in our fine-tuning dataset when answering RQ_1-RQ_4). This is probably not sufficient to effectively train a large model such as T5, and makes the instances used in the pre-training fundamental to further learn about the language. Still, even without pre-training, T5 outperforms the baseline when k=1. For example, in the *code & comment-to-code* task, the baseline achieves 9.48% perfect predictions, against the 15.46% of the non pre-trained T5, and the 29.74% of the pre-trained T5. The baseline observes a stronger improvement with the increasing of k (*i.e.,* the beam size) as compared to T5 (see Fig. 4.4). We believe this is due to usage of the abstraction. Indeed, when working with abstracted code the "search space" (*i.e.,* the number of possible solutions that can be generated with the given vocabulary) is much more limited since the model does not deal with identifiers and literals. Attempting ten predictions in a smaller search space is more likely to result in correct predictions. The results of the CodeBLEU confirm the trend observed with the perfect predictions, with the pre-trained T5 being the best model.

We also looked at the union of perfect predictions generated by the two approaches on the test set to verify the complementarity of the techniques. On the *code-to-code* (*code&comment-to-code*) task we observed that 15% (24%) of perfect predictions are shared by both approaches (*i.e.,* both succeed), 65% (70%) are perfect predictions only for T5, and 20% (6%) only for the baseline.

4.4 Conclusion

The work in this chapter is motivated by the limitations of the approach we presented in Chapter 3. We highlighted that the usage of code abstraction does not allow to support non-trivial code review scenarios requiring code changes resulting in the introduction of new identifiers/literals. Hence, we proposed the usage of a pre-trained T5 model [RSR+20] relying on a SentencePiece [KR18] tokenizer to overcome such a limitation and work directly on raw source code. Our empirical evaluation, performed on a much larger and realistic code review dataset, shows the improvements brought by the T5 model that represents a step forward as compared to the baseline [TPT+21] both in terms of applicability (*i.e.,* scenarios in which it can be applied) and performance.

The obtained improvement is in part due to the pre-training phase that provides the model with general knowledge about the language used in the downstream task. While we relied on the *mask language modeling* (MLM) as pre-training objective (*i.e.,* masking 15% of tokens in the input sentence and asking the model to predict them), it is possible that further benefits may be obtained by adopting different and more specialized pre-training objectives. In the next chapter, we investigate this research question, assessing the impact of different pre-training objectives on the performance of DL models automating several code-related tasks, including the code review ones focus of this thesis.

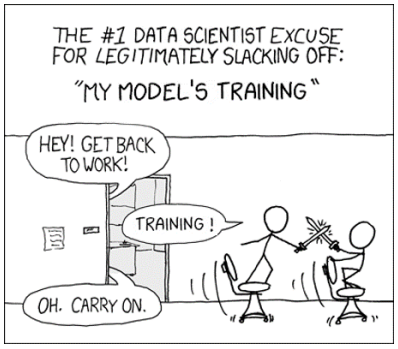
4.5 Replication Package

We release all code and data used in our study in a comprehensive replication package [repd]. It contains:

- the processed and split datasets we used (both for pre-training and fine-tuning);
- the scripts used to preprocess and clean the data;
- the Google Colab notebooks we used to pre-train and fine-tune the models;
- the checkpoints of the best models obtained for each task and dataset, as well as the trained tokenizer model and vocabulary;
- the best models' generated predictions, as well as their scores in terms of BLEU/-code_BLEU, model confidence;
- the results of the manual analysis we performed;
- instructions to replicate our research.

5

Studying the Role of Pre-training on the Automation of Code-related Tasks



The transformer [VSP+17] model we exploit in Chapter 4 to automate code review tasks features a pre-training step using a self-supervised training objective with the goal of providing the model with general knowledge about a language relevant for the final task to automate. The same training procedure has been adopted in the software engineering (SE) literature for the automation of several other code-related tasks (see *e.g.,* [DWSS21, BHRV21, MSC+21, MH21, YKY+21, PLW+21, TLG+21, TMM+22, TPT22a, LLG+22, SDFS20, CCP+21a]).

Most of these works, including ours, adopt as pre-training objective the *Masked Language Model* (MLM), in which a percentage of tokens from the input sentence is masked, with the model in charge of predicting them. Evidence in the literature reports a boost of performance[1] provided by pre-training in the automation of code-related tasks [LLG+22, CCP+21a, MCP+22]. However, little is known about (i) the circumstances in which pre-training actually helps, and (ii) the impact of the specific pre-training objective(s) adopted on the performance of transformers when automating code-related tasks.

Concerning the first point, it is known that pre-training is helpful when the fine-tuning dataset is small [RJ19]. To make a concrete example, per-training can be useful when fine-tuning a model for the *code-to-comment* task. For this task, the fine-tuning dataset is mined from real review activities run in open source projects, thus limiting the amount of training data that can be collected (usually in the order of tens of thousands instances, as in the case of our approach). Pre-training datasets, instead, can be automatically built with virtually "no limitations" in terms of size since the pre-training objective is self-supervised. There are however SE tasks which can leverage very large fine-tuning datasets, for which the boost

[1]With "performance" we refer to the quality of the predictions generated by the model (*e.g.,* accuracy) rather than to attributes such as execution time.

in performance provided by pre-training is not obvious. For example, in the case of code completion, fine-tuning instances are usually represented by pairs of ⟨*incomplete_code*, *complete_code*⟩, with the model learning how to "finalize an implementation task". These pairs can be built automatically from any piece of code by removing parts of it.

As for the second point (*i.e.,* the impact of pre-training objectives), the widely used MLM is just one of the possibilities here: Any self-supervised task can be used as pre-training objective. Also, in a recent work from NLP, Zhang *et al.* hypothesized that "*using a pre-training objective that more closely resembles the downstream task leads to better fine-tuning performance*" [ZZSL20]. We present a large-scale study aimed at (i) investigating whether pre-training is actually useful in code-related tasks for which the fine-tuning dataset can be built without any obvious impediment in collecting large amount of data (*e.g.,* code completion); and (ii) experimenting the impact on transformers' performance of both generic and task-specific pre-training objectives when automating three "generic" code-related tasks, namely *bug-fixing*, *code summarization*, *code completion* and two code review-related tasks, namely *code & comment-to-code*, *code-to-comment*. While the focus of this thesis is on code review, investigating the impact of pre-training on more code-related tasks allows a more generalizable answer to our research question.

We start by performing a systematic literature review to identify the pre-training objectives used in the SE literature. Based on this analysis, we selected three generic pre-training objectives to experiment based on their popularity and potential impact in SE: MLM, *next sentence prediction*, and *replaced tokens detection*. Moreover, we defined four pre-training objectives tailored for the specific downstream tasks we aim at supporting. We then trained 56 T5 models [RSR+20], accounting for a total of 1'390 training hours, using different pre-training objectives and fine-tuning datasets. In particular, we study the impact of the fine-tuning dataset size on the "boost in performance" (if any) provided by pre-training. Also, we assess the impact on the achieved performance of different combinations of pre-training objectives.

Our findings suggest that:

1. Pre-training is extremely useful when the pre-training dataset is substantially larger than the fine-tuning one, while it does not help when the fine-tuning dataset is of comparable size.

2. The MLM pre-training objective represents a safe choice for all tasks we investigated, being almost always the best-in-class; (iii) specialized pre-training objectives only help if they strictly resemble the fine-tuning task and can provide the model with knowledge that cannot be captured by generic objectives.

The content of this chapter has been presented in the following paper:

> **Automating Code-Related Tasks Through Transformers: The Impact of Pre-training**
>
> Rosalia Tufano, Luca Pascarella, Gabriele Bavota. In *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023)*, pp. 2425-2437

5.1 A SLR on Pre-training Objectives Used to Automate Code-Related Tasks

Section 5.1.1 describes the design of our SLR following the guidelines by Kitchenham and Charters [KC07]. The achieved results are discussed in Section 5.1.2.

5.1.1 Study Design

Our SLR aims at answering the following research question: *What are the pre-training objectives used in the SE literature exploiting transformers to automate code-related tasks?*

With "code-related tasks" we refer to any task involving source code as input and/or output of the model. For example, code summarization is considered a code-related task, since it takes as input a code component to summarize in natural language. Differently, using transformers to automate "sentiment analysis" on software-related artifacts (*e.g.,* discussions in issue trackers) is not considered relevant for our SLR. Answering this research question will inform the study presented in Section 5.2, in which we experiment with representative pre-training objectives from the state-of-the-art.

Relevant Study Identification

We used the following digital libraries to search for primary studies: ACM Digital Library [ACM], IEEE Xplore Digital Library [IEE], Springer Link Online Library [Spr], Wiley Online Library [Wil], Elsevier ScienceDirect [Els], and Scopus [Sco]. Google Scholar was not considered as an option due to the lack of quality control, clear indexing guidelines, and missing support for data download [HMBI17b]. The following search query has been run on the search engines integrated in each of these online databases:

full text CONTAINS
("*pretrain*" OR "*pretrained*" OR "*pretraining*" OR
"*pre-train*" OR "*pre-trained*" OR "*pre-training*" OR
"*transfer learning*") AND
publication date IS FROM *01.01.2007* TO *02.02.2022* AND **publication venue** CONTAINS
("*software*" OR "*program*" OR "*code*")

The composition of the query is the result of a trial-and-error procedure performed by three researchers. The query searches for the listed terms (*e.g.,* pretrain, pretrained) in the full text of the articles (*i.e.,* title, keywords, abstract, main text, references). The date interval has been defined by conservatively collecting papers starting from 2007, year in which we found a first mention to the notion of "transfer learning" in a SE-related article [AZFL22], and using the date in which the search has been performed as the end of the interval (02.02.2022). Finally, based on our knowledge of the existing SE publication venues, we only searched for articles published in venues containing at least one of three keywords: software, program, and code. We acknowledge that there might be relevant articles published in related fields (*e.g.,* artificial intelligence) that our query would exclude.

However, our focus was indeed on the SE research community and these keywords should capture most of the relevant venues.

Some of the search engines (*i.e.,* Springer, Wiley, Elsevier, and Scopus) allow to specify a *discipline* of interest. Such a feature is useful to limit the retrieved false positive instances. In all online libraries we selected "Computer Science" as discipline. In addition, Springer also allows to specify sub-disciplines, for which we selected "Software Engineering/Programming" and "Operating Systems". While the latter might not be fully relevant, we decided to include it to be more conservative. Links with the exact queries we have run are publicly available [repe].

Table 5.1. Articles returned by the queried digital libraries

Source	**Returned Articles**
ACM Digital Library	623
IEEE Xplore Digital Library	850
Springer Link Online Library	1,167
Wiley Online Library	57
Elsevier ScienceDirect	288
Scopus	1,139
Total (including duplicates)	4,124
Total (excluding duplicates)	**2,343**

Table 5.1 reports the number of articles returned from each digital library (complete list in [repe]). Overall, 4,124 articles were returned, which were reduced to 2,343 by excluding duplicates. Given the very high number of articles, we decided to perform a further cleaning step before starting looking into the papers. We extracted the set of 302 venues in which the articles have been published and two researchers independently validated them deciding whether to include or exclude them. We excluded venues unrelated to SE or not being international conferences/journals. An open discussion was performed to reach an agreement on the 53 cases of conflict (17%). As output of this process we kept 163 publication venues as valid, excluding 1,407 papers published in the excluded venues. Examples of excluded publication venues are "*Computer Methods and Programs in Biomedicine*" and the "*Brazilian Symposium on Programming Languages*". As output we obtained 936 candidate primary studies.

Study Filtering. The 936 papers were equally distributed among the three involved researchers. Each of them was in charge of inspecting the paper and decide whether to include or exclude it. Inclusion and exclusion criteria are listed in Table 5.2. As a guideline, we agreed on including the paper in case of doubt, since a double-check was foreseen in the study filtering process. Indeed, despite the availability of the selection criteria as reference, such a process still remains highly subjective. A total of 77 papers survived this first analysis. Then, to at least partially address the subjectivity issue, we applied the following procedure.

Table 5.2. Inclusion and exclusion criteria

	Inclusion Criteria
IC1	The paper must be peer-reviewed, published at SE conferences, workshops, or journals. Such a criterion is particularly important in the snowballing phase described later, in which we ignore all referenced preprints (*e.g.,* those published on arXiv.org).
IC2	The PDF of the paper must be available online. If the PDF was not available in the online library of interest, we tried to search it on Google.
IC3	The paper must present and/or evaluate technique(s) to automate a code-related task.
IC4	The proposed/experimented technique(s) must be built on top of a pre-trained transformers model. The pre-training of the model can either have been done directly in the paper or the authors may have used an already existing pre-trained model.
	Exclusion Criteria
EC1	The paper is not written in English.
EC2	The paper has been published in a conference/workshop and later on extended to a journal. We only keep the journal paper to avoid redundancy.
EC3	The paper is not a full research publication (*e.g.,* doctoral symposium articles, posters, ERA track). We exclude all papers having less than six pages. The rationale for such a filter is to remove papers that may not have been subject to the same peer-review process typical of full research papers.
EC4	It is unclear from the paper what the adopted pre-training objective is. Such information is instrumental for the goal of our SLR.

First, we randomly selected 30 papers excluded by each researcher, asking one of the other two researchers to double check whether the papers were actually to be excluded. For all 90 randomly selected papers (30 × 3 researchers), no conflicts arisen, showing consistency in the exclusion criteria applied. The papers included by each researcher were also all double-checked by one of the other two researchers. Out of the 77 papers included in the first round, 30 made it into the final list of papers, including a SLR that we kept as secondary study for the subsequent snowballing step.

Cases of disagreement have been discussed among all researchers to reach consensus. Note that the decrease brought by the double-check we performed (77 → 30) was expected, considering that in the first pass on the papers we decided to be inclusive in case of doubts.

Backward Snowballing. The included papers were split among the three researchers, with each of them in charge of reading the reference list and identify possible relevant papers. At this stage we relaxed one of our inclusion criteria (IC1): We agreed to include papers published in venues outside of SE as long as they were presenting pre-trained models that have then been exploited to automate code-related tasks in publications appeared in SE venues. Eight papers were added through snowballing. Also in this case, a double-check was performed on each of them and, through open discussion among all researchers, we finally agreed to include four of them as relevant. This led to the final list of 33 primary studies included (30 - 1 secondary study + 4 output of the snowballing).

Table 5.3. Data extraction questionnaire

No.	Question
1	Which code-related task has been automated?
2	Which specific transformer-based model has been used?
3	Has the model been pre-training in the paper?
4	If "no" to question 3: Which already pre-trained model has been exploited by the authors?
5	Which pre-training objectives have been used?

Data Extraction and Analysis

The data extraction was performed following the questionnaire in Table 5.3. While most of the questions are self-explanatory, it is worthwhile to clarify 3 and 4. Our focus is only on papers using pre-trained transformers to automate code-related tasks. However, the pre-training could have been done by the authors of the papers or being the result of a pre-trained model made available in previous work (question 3). If the authors reused an already pre-trained model, then by answering question 4 we expect to know from which reference the pre-trained model has been taken. Question 5 is the ultimate goal of our SLR, which will inform our subsequent experiments.

5.1.2 Results Discussion

Table 5.4 lists the pre-training objectives we identified. Each pre-training objective is identified by an acronym we will use to refer to it in the text. If the acronym has a </> icon close to it, this indicates that the pre-training task is specific for code, otherwise the pre-training objective is "generic" and can be applied to any sort of data that can be fed to the model as a stream of tokens. For each pre-training objective Table 5.4 also reports a short description and references to primary studies in the SLR having a pre-trained model using it.

Without surprise, the most used pre-training objectives are those that the SE community inherited from the NLP community, such as the classic MLM, randomly masking X% of tokens in an instance that, in the case of SE research, could be for example a code function. MLM is used in 21 of the papers included in our SLR. Variations of this pre-training objective are TI, ULM, TD, and RTD (see Table 5.4 for their description) that are, however, less popular in SE. Among them, the *Replaced Token Detection* (RTD) objective has been proposed in the paper introducing the pre-trained CodeBERT model [FGT+20]. CodeBERT is gaining substantial popularity in the SE literature especially when it comes to papers published in 2022 that, due to the time in which we run the search query, are not included in our SLR.

While the above-described objectives work at token-level granularity, *Next Sentence Prediction* (NSP) and *Sentence Ordering* (SO) aim at providing the model with knowledge related to sentence-level relationships. In SE, both can be used for example to "teach" the model the correct order of code statements in a given function. NSP is the second most-popular pre-training objective in our set of papers, with 9 articles exploiting it. The popularity of NSP is mostly due to the adoption of the BERT pre-trained model [HS97] in SE.

Table 5.4. Pre-training objectives identified in the SLR

Name	Description	References
Masked Language Model (MLM)	Masks *X*% of tokens (usually 15%) in the instance (*e.g.,* a function) and asks the model to guess the masked tokens based on their bidirectional context. The model knows how many tokens have been masked, since each of them is replaced with a special token (*e.g.,* <MASK>).	[ADS22, SDFS20, ZMW19, CR21, ZPW+21, HMS20, QCY21, GXL+21, LPM+21, MMPT21, GJMM21, CCP+21b, MSC+21, LLZ+21, MAPB21, SXP+21, ZLL+21, ZHL21, GRL+21, ACRC21]
Next Sentence Prediction (NSP)	Given two sentences (or two statements) asks the model to guess whether they follow each other.	[ZMW19, HST+21, GXL+21, LPM+21, MMPT21, LLZ+21, SXP+21, ZLL+21]
Unidirectional Language Model (ULM)	A left-to-right language modeling task, asking the model to guess one masked token in an instance by only considering the leftward tokens (*i.e.,* the tokens preceding the masked one).	[LLZJ20, HHJC21, JLT21]
Token Infilling (TI)	Masks a random number of contiguous tokens and asks the model to predict them. Differently from MLM, TI does not suggest to the model how many tokens have been masked, since the sequence of masked tokens is replaced with a single special token (*e.g.,* <MASK>).	[CR21]
Token Deletion (TD)	Deletes random tokens from the instance expecting the model to reintroduce them where needed. TD is similar to MLM, but without suggesting the model where tokens have been masked.	[CR21]
Replaced Tokens Detection (RTD)	Replaces random tokens in the instance with other tokens. The model must guess which are the non-original tokens (*i.e.,* those that have been replaced).	[ZPW+21]
Sentence Ordering (SO)	Given two sentences (or two statements) asks the model to guess whether they order.	[ATLJ20]
Identifiers Masking (IM </>)	Masks the identifiers in the code instance and asks the model to guess the masked identifiers.	[LLZJ20]
Programming Language Classification (PLC </>)	Given a sequence of code tokens asks the model to identify its programming language.	[SDFS20]
Generative State Modeling (GSM </>)	Given assembly code and a small subset of its execution states (*e.g.,* register values), asks the model to reconstruct the complete set of its execution states.	[PGB+21]
Edge Prediction (EP </>)	Masks edges in data-flow graph belonging to 20% of nodes randomly selected and asks the model to predict them.	[GRL+21]
Node Alignment (NA </>)	Similar to data flow EP. Instead of predicting edges between nodes, the model is asked to predict edges between code tokens and nodes. Such a task is performed to align the source code-data flow representations.	[GRL+21]
Code Summarization (CS </>)	Provides as input to the model a function and asks to summarize it in natural language.	[WLX+19]

The bottom of Table 5.4 lists the objectives specifically designed for code. Despite being pre-training objectives, some of them are targeting a specific task, like for example *Code Summarization* (CS). The latter has been instantiated in [YKY+21] as a task in which the model was asked, given a Java method, to generate its textual summary (*i.e.,* first Javadoc sentence).

The final code-related task that the authors wanted to automate after the fine-tuning phase was a natural language description of smart contract. Thus, the CS objective started providing the model with knowledge about the code-to-NL translation task. This is a concrete example of pre-training objective already tailored for the specific downstream task of interest. Overall, as it can be seen from Table 5.4, code-specific pre-training objectives have been usually adopted only in the paper proposing them.

Section 5.2 describes how we use the findings of our SLR to select the pre-training objectives for our experiments.

5.2 Studying the Impact of Pre-training

We aim at answering the following research questions:

RQ_1: *To what extent is the effectiveness of pre-training influenced by the size of the fine-tuning dataset?* RQ_1 investigates if pre-training is still useful when abundant fine-tuning instances are available (*e.g.,* can be automatically generated). We assess the impact of pre-training when the model is fine-tuned on a dataset (i) being substantially smaller and (ii) having a size comparable to the pre-training dataset.

RQ_2: *To what extent does the choice of the pre-training objective impact the performance of transformer models?* Our second research question focuses on the impact on the model's performance of the used pre-training objectives, with a particular focus on comparing general *vs* task-specific objectives, also investigating combinations of multiple objectives.

The context of our experiments are (i) three code-related tasks for which DL-based solutions have been proposed in the past, and (ii) two code review-related tasks, namely *code & comment-to-code* and *code-to-comment*. For reasons we will explain later, RQ_1 focuses on *code summarization* and *code completion* only, while RQ_2 includes all tasks. All our experiments are run on Java datasets defined at method-level granularity (*e.g.,* fixing a bug in a method).

5.2.1 Transformer Model

As representative of transformers [VSP+17], we adopt the T5 proposed by Raffel *et al.* [RSR+20], that has been widely used in SE to automate code-related tasks. Raffel *et al.* proposed several variants of T5, differing in number of trainable parameters.

We use the *small* variant, featuring a total of ∼60M parameters resulting from 6 layers in both the encoder and the decoder each having a dimensionality of 512, 8-headed attention, and an output dimensionality of 2,048 (same used in our work summarized in Chapter 4). While larger T5 versions are likely to achieve better performance, the training cost increases with the number of parameters. Considering the number of models that we need to train in our study (*i.e.,* 56 different models), we opted for the smaller T5 version.

Indeed, our goal is not to achieve state-of-the-art performance in the automated tasks, but rather to study the impact of pre-training on the model's performance in different circumstances.

5.2.2 Pre-training Objectives

We describe the pre-training objectives used in our RQs.

Concerning RQ_1, we only use the *Masked Language Model* (MLM) objective, the one mostly used in the literature. Indeed, the focus of RQ_1 is not on the impact of different pre-training objectives (RQ_2), but rather on the boost provided by pre-training when the fine-tuning dataset is substantially smaller than the pre-training one or, instead, has a comparable size.

For RQ_2, based on the findings of our SLR, we selected three generic pre-training objectives to experiment with.

First, we picked the two currently being the most popular in SE, namely the MLM (with 15% of masked tokens) and the *Next Sentence Prediction* (NSP). As third "generic" objective, we selected the *Replaced Tokens Detection* (RTD) that, as previously explained, is gaining popularity since used in CodeBERT [FGT+20], adopted in several recent studies (see *e.g.,* [MH21, ZYK+22, PLX21]). While MLM and NSP are straightforward to understand, a clarification is needed on RTD. The latter starts by randomly selecting 15% of tokens to be replaced. However, the replacements for these tokens are not randomly selected from a vocabulary, but picked based on the recommendation of an n-gram model ($n = 3$) trained on the pre-training dataset. An n-gram model can predict a single token likely to follow the $n-1$ tokens preceding it. As suggested in [FGT+20], we used it to identify suitable alternatives for the tokens to be replaced, thus making the pre-training task (*i.e.,* identify which tokens in an instance have been replaced) more challenging. Given a token to replace T_i, we run the n-gram model by providing it as input with the two tokens preceding it (T_{i-2}, T_{i-1}) and collect the ranked list of candidate tokens that, accordingly to the n-gram model, is likely to follow T_{i-2} and T_{i-1}. The ranked list features on top the most likely token T_c: If T_c is different from the token to replace T_i, we use T_c for the replacement. Otherwise, we take the token in second position.

On top of the three generic objectives, we also experiment with four pre-training objectives tailored for the downstream tasks at hand.

For ***bug-fixing***, we pre-train the model through the *Injected-Mutants Fixing* (IMF) objective. The idea is to mutate each method M in the pre-training dataset by injecting artificial bugs in it, creating a mutant M_m. During pre-training the IMF objective provides T5 with M_m as input and asks it to generate M (*i.e.,* to fix the bug). One challenge we faced was the selection of the mutation testing framework to use. We considered tools such as *μJava* [muj], *PIT* [pit], *javaLanche* [javb], and *Jester* [Jes]. PIT was the only one supporting recent versions of Java. However, since it works at Byte code level, PIT requires the input code to mutate to be compilable. This is problematic in our context since the 1M methods in our pre-training dataset come from thousands of software projects, several of which are likely to be unbuildable [TPB+17].

For this reason, we built a source code-level mutation tool using Javaparser [java]. Our tool implements the 11 mutation operators belonging to the "default group" in PIT [opeb] (*i.e., invert negatives*, *empty returns*, etc.). Given a Java method M, our tool builds its AST and, using it, identifies the set of mutation operators that can be applied on M. For example, the *empty returns* operator replaces return values with "empty" values (*e.g.,* `""` if M returns a `String`, `Collections.emnptyList()` if M returns a `Collection`, etc.), and can only be applied to methods returning a value. Finally, assuming that n operators can be applied to M, n mutants (*i.e., n* versions of M) are generated, each implementing one of the applicable operators.

Concerning the ***code summarization*** task, we consider the *Method Name Generation* (MNG) as a tailored pre-training objective. During pre-training, T5 takes as input a Java method and it is required to synthesize an appropriate name for it, based on the idea that the method name represents an *extreme summary* of the method [APS16].

Regarding ***code completion***, we focus on the challenging task of predicting entire code blocks, as recently attempted by Ciniselli *et al.* [CCP+21a]. A code block is defined as the code enclosed between two curly brackets (*e.g.,* the code executed when an `if/else/else if` condition is satisfied). To prepare the model for such a downstream task, we devised *Code Block Selection* (CBS) as a tailored pre-training objective. Given a Java method in the pre-training dataset, we randomly mask a code block in it, and ask the model to decide which of two candidate code blocks is the correct one to complete the method. This pre-training is expected to prepare the model for the more challenging downstream task of generating masked code blocks from scratch.

Finally, for the two code review-related tasks (*i.e.,* ***code & comment-to-code, code-to-comment***) we defined as pre-training objective the *Revised Code Lines Prediction* (RCLP), namely providing the model with the code submitted for review and asking it to guess which lines will be impacted by changes introduced in the review process. Indeed, both downstream code-review tasks we aim at automating require the model to understand which parts of the code must be revised. For the *code & comment-to-code* task the model outputs the revised code having as context the code to be revised and the reviewer comment. Thus, to succeed in the task, the model has to correctly identify the code lines to be modified (and then revise them). Regarding the *code-to-comment* task, the model is required to generate a code change request in natural language as a reviewer would do. Also in this case, to succeed, the model needs to identify which type of code change is necessary for the given code, thus, which lines of code are required to be changed. For these reasons, predicting the code lines to change as a pre-training objective may help the model in the code review-related downstream tasks. Formally, given a Java method as input, we ask the model to highlight (using special tokens) the code lines that would be revised. As described in the next section, due to its different nature, for this pre-training objective we will use a different pre-training dataset as compared to the other three code-related tasks.

5.2.3 Pre-training Datasets

We built two different pre-training datasets: one for code-related tasks and one for code review-related tasks. The reason behind this decision is given by the specific nature of the instances needed for the RCLP pre-training objective, which are output of a code review process, differently from the other three tasks. Also, the fine-tuning dataset for the code review tasks also exploits instances output of the code review process, thus requiring to split the available instances between pre-training and fine-tuning. As a consequence, the pre-training dataset available for the code review tasks will be quite limited in terms of size as compared to the other tasks.

Code-related Pre-training Dataset

To build the pre-training dataset for the code-related tasks, we used the GitHub search tool by Dabić *et al.* [DAB21, ghs] to identify GitHub Java projects having at least 5 contributors, 50 commits, 10 stars and not being forks of other projects. These selection criteria aimed at removing personal projects from our selection. We ended up with 14,645 valid repositories, that we parsed to extract 64,546,432 Java methods.

Since among the downstream tasks we experiment with there is *code summarization*, we wanted to make sure that each pre-training instance was composed by both source code (instrumental for all three tasks) and natural language (useful for *code summarization*). Also, recent studies [TDS+20] showed that pre-training models on both natural language and code (as opposed to code only) is beneficial when dealing with code-related tasks. For these reasons, methods without Javadoc have been excluded, leading to 17,758,579 $\langle method, javadoc \rangle$ pairs.

We then started processing our dataset to clean it and remove problematic instances. We excluded all pairs meeting one of the following conditions: (i) the Javadoc, while present, is an empty string; (ii) the method has an empty body; (iii) the method is annotated with `@Test`; (iv) the method does not end with a `}` (this may happen in case of parsing errors when we extract the methods). We excluded test methods since none of our tasks is test-related, and we preferred to create a more cohesive pre-training dataset featuring only methods from production code. We only consider in our dataset the first part of the Javadoc comment (*i.e.,* the one summarizing the method in natural language) excluding the Javadoc tags (*e.g.,* `@param`, `@author`). Once done with this basic filtering, we manually inspected hundreds of instances in the dataset to identify other sources of noise.

Four main issues were identified: (i) non-English Javadoc comments; (ii) instances containing non-ASCII characters; (iii) comments containing special symbols/tags which may not help with learning textual patterns in Javadoc; and (iv) comments not representing code summaries, but rather notes written by developers (*e.g.,* TODOs). We removed all non-English Javadocs using two Python libraries: *langid* [lanb] and *cld3* [pyc]. We keep an instance in the dataset only if both libraries classified the Javadoc as English text. We replaced all non-ASCII math characters with their corresponding ASCII representation (*e.g.,* we replaced "±", with "+-") and removed all instances featuring non-Latin characters.

Additional cleaning aimed at removing from the Javadoc sequences of characters used

for formatting (*e.g.,* "`---`") or special markdown tags (*e.g.,* we replace `{@class ClassName}` with `ClassName`). We also replace any embedded link in the Java method and/or in the Javadoc with a special tag "`<LINK_i>`", with i being an integer ranging from 0 to $n-1$, where n is the number of links in the instance. If the same link is found both in the method and in its Javadoc, they are replaced with the same special tag with the same index. Finally, since the collected methods came from different projects possibly using different coding styles, we formatted all instances using the Javaparser [java] library. We also performed additional (minor) cleaning steps that we do not document here. However, we publicly release our cleaning script as part of our replication package [repe].

After this process, we removed all instances longer than 512 tokens (*i.e.,* the number of tokens used to represent both the method and its Javadoc was higher than 512), as also done by previous work using DL to automate code-related tasks (see *e.g.,* [LM19, TDS+20, CKT+21]): 4,821,922 instances were left.

As a last step, we excluded instances that are not suitable for one or more of our pre-training objectives defined for the three code-related tasks. The two objectives which are not applicable to all possible instances are the *Injected Mutants Fixing* (IMF) and the *Code Block Selection* (CBS): We removed (i) 219,863 instances for which none of the 11 mutation operators we support can be applied; and (ii) 2,994,723 which did not have any code block to mask. From the set of remaining instances, we randomly pick 1M of them to create our pre-training dataset for the code-related tasks. While, in theory, all ∼1.6M remaining instances were valid, we capped the size of the pre-training dataset to limit the time needed to perform several training epochs.

Code Review-related Pre-training Dataset

To build the pre-training dataset for the code review-related tasks, we use Megadiff [MMY+21], a collection of 600k Java code changes organized by diff size, ranging from 1 to 40 lines of code changes. The goal of the model when solving the RCLP pre-training objective is to correctly guess which lines of code will be revised, highlighting them using special tokens(`<rev>, </rev>`); considering 40 lines of code changes means that the model must correctly identify all of them to succeed and this could lead to an overly demanding pre-training task. For this reason, we select only data points featuring between 1 and 15 lines of code changes.

Each instance in this dataset consists of an entire file diff obtained after a commit operation. Since our granularity focus is method level, we parsed the file extracting the impacted methods (*i.e.,* those featuring at least one changed line). Then, from the 256,211 collected methods we filtered out: (i) methods with an empty body; (ii) constructor methods; (iii) methods annotated with `@Test`; and (iv) methods that do not end with a `}` (likely the result of a parsing error). Finally, we removed all instances longer than 512 tokens. This process left us with 115,346 valid instances. Since the considered code review-related tasks need the model to understand natural language (to implement the requested code change in *code & comment-to-code* and to correctly generate the comment in *code-to-comment*) in this dataset we do not remove inline comments.

This is done because, differently from the pre-training dataset used for the three code-related tasks, in this dataset there are very few methods having a Javadoc. We removed non-English comments and cleaned the valid ones by removing punctuation symbols (*e.g.,*'*', '//', '/*'). Finally, since the instances will be flattened to be suitable for the model (*i.e.,* no code indentation will be available), we use special tokens (`<technical_language>`, `</technical_language>`) to mark the beginning and end of comments inside the code.

As a last step, we excluded instances that are not suitable for the pre-training objective defined for the three code review-related tasks, *i.e., Revised Code Lines Prediction* (RCLP). We removed 11'543 instances consisting of methods where the changed lines of code are just added lines. We do this because in those cases it is not possible to highlight the lines of code that will be modified. The obtained dataset counts 103,803 instances.

5.2.4 Fine-tuning Datasets

A different fine-tuning dataset has been built for each of the five subject downstream tasks. Due to their limited size, the bug-fixing dataset and the code-review datasets have only be used in the context of RQ_2, since it was not possible to create a version of them large enough to answer RQ_1 as well.

Bug-fixing

We exploit the dataset used by Chen *et al.* [CKT+21] when presenting SequenceR, a sequence-to-sequence model trained on 35,578 one-line Java bug fixes (*i.e.,* commits fixing a bug by only changing a single line of code). The training set consists of pairs featuring the buggy code and the corresponding fixed code, and it is accompanied by a validation and a testing set featuring additional 4,711 one-line bug fixes each. The buggy code includes a "buggy line" explicitly marked with two special tokens (*i.e.,* ⟨START_BUG⟩ and ⟨END_BUG⟩) and being part of a "buggy method". In addition to that, the buggy code also includes contextual information extracted from the "buggy class" (*e.g.,* its constructor). The fixed code the model is expected to generate includes, instead, only the "fixed line" (*i.e.,* revised version of the "buggy line").

Before using this dataset, we pre-processed it to make it more "aligned" to our pre-training dataset, and in particular to the tailored pre-training objective we devised for the *bug-fixing* task (*i.e., Injected-Mutants Fixing*). First, our pre-training dataset only features Java methods with its related Javadoc, excluding any class-related information. Similarly, the IMF objective provides as input to the model a mutated Java method without any additional contextual information nor special tag signaling the injected bug. Thus, we processed the buggy code of the dataset by Chen *et al.* [CKT+21] to only include the buggy method without the special tokens marking the buggy line. Second, as done for the pre-training instances, we formatted the code using Javaparser [java], to have a coherent code representation. Finally, we removed any duplicated method already present in our pre-training dataset. This process left us with 25,901 instances that we split into training (80%), validation (10%) and test (10%) set.

Code summarization

We use the FunCom dataset [LM19, fun], featuring 2,149,120 instances, with each of them being composed by a Java method and its associated Javadoc comment. FunCom has been curated to only include English comments and exclude auto-generated files. We start from the "Filtered dataset" version [fun] consisting of not processed instances. We perform on them the same cleaning process used for the pre-training dataset (*e.g.,* removing instances containing non-Latin characters) and remove any duplicate with between FunCom and our pre-training dataset. From the 1,898,437 instances left, we create two fine-tuning datasets needed to answer RQ_1. For the first (*large-ft*), we randomly select 1M instances, splitting them into training (80%), validation (10%), and test (10%). This fine-tuning dataset will be used in RQ_1 as representative of a fine-tuning dataset having a size of the same order of magnitude of the pre-training dataset. For the second (*small-ft*), we randomly select 25,901 instances, the same number of instances in our *bug-fixing* fine-tuning dataset. The idea is indeed to create a second fine-tuning dataset being substantially smaller than the pre-training one, as it usually happens when working on tasks characterized by a scarcity of training data. Also *small-ft* followed the usual training (80%), validation (10%), and test (10%) split. Both datasets are used to answer RQ_1, while only *small-ft* is used in RQ_2. Indeed, as our RQ_1's findings will show, pre-training is mostly useful when a small fine-tuning dataset is available. Thus, we experiment the impact on performance of the pre-training objectives (*i.e.,* RQ_2) when using the *small-ft* dataset.

Block-level code completion

Following Ciniselli *et al.* [CCP+21a], we aim at building a fine-tuning dataset in which Java methods having a masked block of up to three statements are provided as input to T5, which is in charge of generating the masked block. We start from the 1,569,889 Java methods in the CodeSearchNet dataset [HWG+19]. We applied a cleaning process similar to the one described for the pre-training dataset (*e.g.,* checking that the method body is not empty, that the method is not a test method, etc.), removed methods not containing at least one code block composed by at most three code statements (2,847).

Then, we followed the training procedure by Ciniselli *et al.* [CCP+21a]: Given k the number of blocks identified in a method M, we create k versions of M, each one having a specific code block masked. As previously said, only blocks composed by at most three statements are masked. We then remove instances longer than 512 tokens (333,955). Such a process resulted in a dataset composed by 1,823,977 instances. Finally, similarly to what explained for code summarization, we create two versions of the *code completion* fine-tuning dataset: *large-ft* featuring a total of 1M randomly selected instances, and *small-ft* featuring 25,901 instances.

Code Review dataset

Regarding the two code review related-tasks we use the datasets used in [TMM+22] (Section 4.1.2). They are two datasets consisting of: (i) triplets $< m_s, r_{nl}, m_r >$, where m_s is a method submitted for the review, r_{nl} is a single reviewer's comment suggesting code changes for m_s, and m_r is the revised version of m_s implementing the reviewer's recommendation r_{nl} (*code & comment-to-code*); (ii) pairs $< m_s, r_{nl} >$ where m_s is a method submitted for the review and r_{nl} is a single reviewer's comment suggesting code changes for m_s (*code-to-comment*). Note that there is no overlap between the instances in these fine-tuning datasets and those in the pre-training dataset we built for the code review tasks.

Table 5.5 summarizes the pre-training/fine-tuning datasets, also indicating which dataset has been used in each RQ.

Table 5.5. Pre-training and fine-tuning datasets used in our study

Dataset	Training	Evaluation	Test	RQ_1	RQ_2
Pre-training					
code-related tasks	1,000,000	-	-	✓	✓
code review-related tasks	103,803	-	-	✗	✓
Fine-tuning					
Bug-fixing	22,321	2,790	2,790	✗	✓
Code summarization					
large-ft	800,000	100,000	100,000	✓	✗
small-ft	22,321	2,790	2,790	✓	✓
Code completion					
large-ft	800,000	100,000	100,000	✓	✗
small-ft	22,321	2,790	2,790	✓	✓
Code&comment-to-code	134,239	16,780	16,780	✗	✓
Code-to-comment	134,239	16,780	16,780	✗	✓

5.2.5 Experimental Procedure

The training of the models has been performed using a 2x2 TPU topology (8 cores) from Google Colab with a batch size of 128 and the *Inverse Square Root* learning rate.

Answering RQ_1

We start by fine-tuning (without pre-training) four models, two for the *code summarization* and two for the *code completion* task.

The models trained within each task differ for the fine-tuning dataset used, being either the *large-ft* (800k training instances) or the *small-ft* (∼22.3k). The fine-tuning has been performed using an early-stopping training strategy by exploiting the evaluation set. In particular, we saved a checkpoint of the model every epoch computing its performance in terms of correct predictions on the evaluation set and stopped the training if the performance

of the model did not increase for three consecutive checkpoints (to avoid overfitting). With "correct predictions" we refer to cases in which the generated prediction is identical to the target. For code summarization, a correct prediction implies that the summary generated by the model is equal to the one written by developers. In the case code completion, the predicted code block matches the one we masked.

Then, we pre-train a T5 model for 40 epochs on the 1M instances featured in the pre-training dataset using the MLM objective. We fine-tune four versions of it, again two for each task (*i.e., code summarization* and *code completion*) differing for the used fine-tuning dataset (*large-ft* or *small-ft*). We used the same early-stopping procedure described above.

This process resulted in an overall of eight models, four being pre-trained and four not being pre-trained. These eight models have been run on the corresponding test sets collecting their predictions. This allows to compare the pre-trained and the not pre-trained models both when using a fine-tuning dataset having a size comparable to that of the pre-training (*large-ft*) or being substantially smaller than it (*small-ft*).

Answering RQ$_2$

We experiment with four possible pre-training objectives (and their combinations) for each of the tasks subject of our study: three "generic" pre-training objectives (*i.e.,* MLM, NSP, and RTD) that can be applied to any task and the pre-training objective specifically tailored for the given task (*e.g., Method Name Generation* for code summarization). Thus, for each task, we start by pre-training and fine-tuning four T5 models, each one using a specific pre-training objective. All the pre-trainings are run for 40 training epochs, while concerning the fine-tuning, we adopt the same early-stopping training strategy described for RQ$_1$. The above-described process results in 20 different models being pre-trained and fine-tuned (4 $\times$ 5 tasks).

For each task, we evaluate the four fine-tuned models on the corresponding test set (see Table 5.5) in terms of correct predictions, identifying the best performing pre-training objective pt_o. The latter has then been combined in pairs with the remaining three objectives. For example, assuming that *Method Name Generation* (MNG) results the best pre-training objective among the four experimented for the *code summarization* task, we create three pairs of pre-training objectives including <MNG, MLM>, <MNG, NSP>, and <MNG, RTD>. This provides us with additional three models pre-trained and fine-tuned for each task (15 models overall — 3 $\times$ 5 tasks). Again, each of these models has then be evaluated on the corresponding test set, identifying the best performing "pair" of pre-training objectives for each task. The latter has been used to generate two triplets of pre-training objectives by combining it with the remaining two objectives. For example, assuming <MNG, NSP> to be the best pair for *code summarization*, we create <MNG, NSP, MLM>, and <MNG, NSP, RTD>. This results in the training and testing of two additional models for each task (10 overall). Finally, we test for each task the full combination of the four corresponding pre-training objectives, thus training one additional model for each task (5 overall). In total, we pre-trained and fine-tuned 20+15+10+5=50 T5 models in RQ$_2$.

The output of such a process is, for each task, the set of predictions generated by the 10 models trained for it (4 using a single pre-training objective, 3 using pairs of objectives, 2 using triplets, and 1 using all four objectives).

5.2.6 Data Analysis

Using the generated predictions, in both our RQs we assess the performance of the models by computing the percentage of correct predictions generated (*i.e.,* predicted output identical to the expected one). We statistically compare the results achieved by the different models for each task using the McNemar's test [McN47], which is a proportion test suitable to pairwise compare dichotomous results of two different treatments. To account for running multiple test instances (*e.g.,* in RQ_2 comparing the results of the model pre-trained with MLM with those pre-trained using RTD, NSP, etc.), we adjust *p*-values using the Holm's correction [Hol79a]. We complement the McNemar's test with the Odds Ratio (OR) effect size.

Table 5.6. RQ_1: Impact of pre-training when the fine-tuning dataset size (|FT|) is $<<$ or $\sim$ to the pre-training dataset size (|PT|)

|FT| *vs* |PT|	**Task**	**Non Pre-trained**	**Pre-trained (MLM)**
$<<$	Code Summarization	1.94%	4.73%
	Code Completion	2.37%	5.05%
$\sim$	Code Summarization	16.60%	15.98%
	Code Completion	30.41%	29.11%

5.3 Results Discussion

We discuss the achieved results by research question. We highlight the main take-aways of our study using the 💡 icon.

5.3.1 RQ_1: Effectiveness of pre-training when dealing with fine-tuning datasets of different sizes

Table 5.6 reports the percentage of correct predictions achieved by the non pre-trained T5 and the T5 pre-trained using the MLM objective. Results are reported for the two tasks involved in RQ_1 (*i.e., code summarization* and *code completion*) in the scenario in which the fine-tuning dataset is (i) substantially smaller (22.3k $<<$ 1M) than the pre-training dataset (*i.e.,* the *small-ft* dataset has been used — top part of Table 5.6), and (ii) of a size similar (800k $\sim$ 1M) to the pre-training dataset (*i.e.,* the *large-ft* dataset has been used — bottom part of Table 5.6).

💡 When the fine-tuning dataset is small, the pre-training, as expected, helps the learning of the model. For *code summarization*, the boost in terms of perfect predictions goes from 1.94% up to 4.73%, resulting in a statistically significant difference (p-value < 0.001) with

an OR=10.3, indicating ten times higher odds of obtaining a correct prediction from the pre-trained model as compared to the non pre-trained one. Similar findings hold for the *code completion* task, with correct predictions growing from 2.37% to 5.05% thanks to the pre-training (p-value < 0.001, OR=11.7).

Moving to the bottom part of Table 5.6, two observations can be made. First, with the fine-tuning dataset being 36 times larger (22.3k *vs* 800k) the performance of the model improves dramatically both for the pre-trained and for the non pre-trained model. This is kind of expected considering the larger amount of data from which the model can learn useful patterns.

Second, 💡 when the fine-tuning dataset size is similar to that of the pre-training dataset, we do not observe any boost provided by pre-training, with performance slightly in favor of the non pre-trained model in both tasks (p-value < 0.001, OR=1.2 for *code summarization*, and p-value < 0.001, OR=1.3 for *code completion*). While such a result may look surprising, it might be partially explained by the well-known "catastrophic forgetting" phenomenon affecting neural networks [Rob95]: DL models tend to forget previously learned information once new information is provided. Having a large fine-tuning dataset may lead to "override" what the model learned during pre-training, making the latter basically useless.

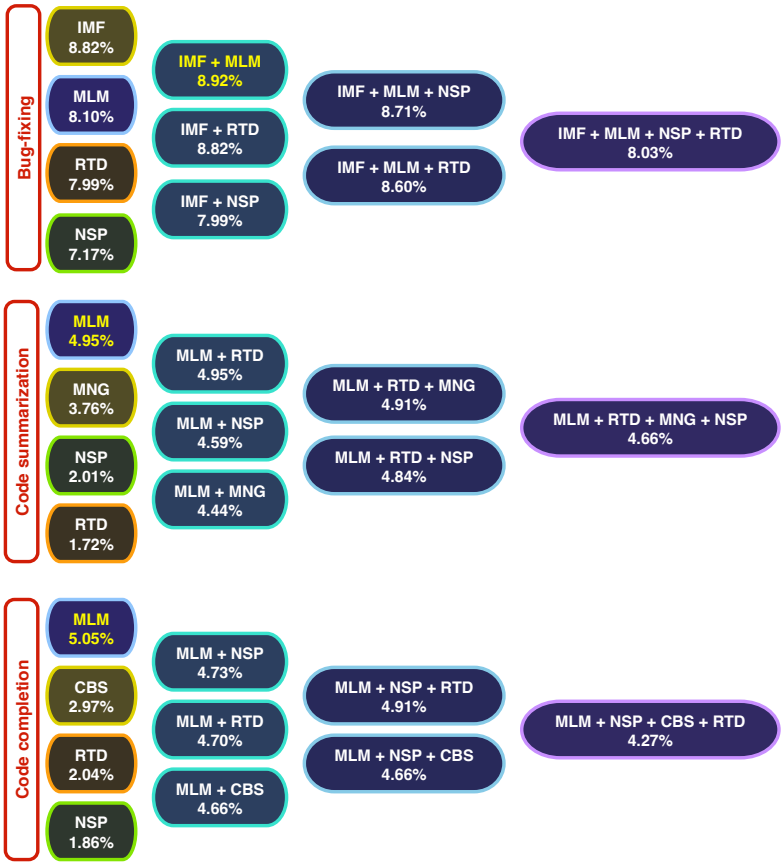


Figure 5.1. RQ_2: Results with different combinations of pre-training objectives for code-related tasks

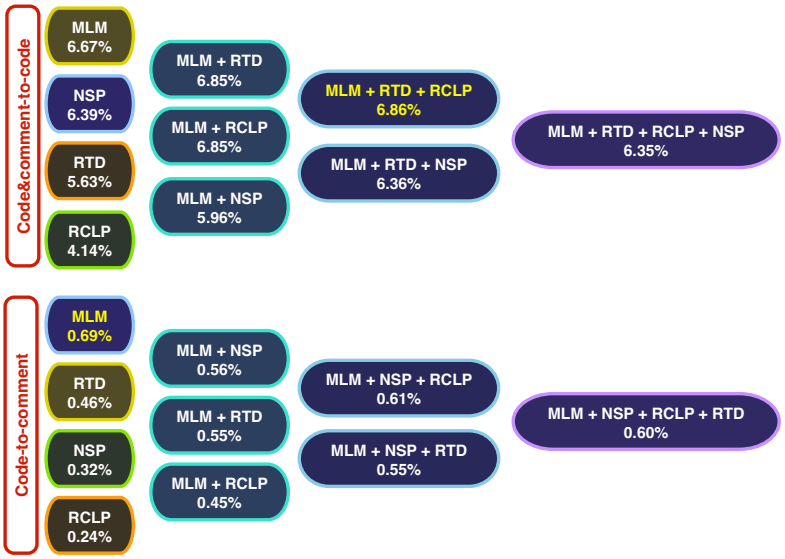


Figure 5.2. RQ_2: Results with different combinations of pre-training objectives for code review-related tasks

5.3.2 RQ_2: Impact of pre-training objectives on performance

Figures 5.1 (generic code-related tasks) and 5.2 (code review tasks) summarize the results in terms of correct predictions achieved by the 50 models pre-trained using different objectives (10 for each task).

From the left to the right of the two figures we move from pre-trainings performed with a single objective (*e.g.,* MLM) to those involving all objectives relevant for a given task (*e.g.,* IMF + MLM + NSP + RTD). Within each task and each pre-training group (*i.e.,* those involving only one objective, those involving two objectives, etc.), the objectives are sorted from the top to the bottom based on the performance they ensured.

For example, for *bug-fixing*, when experimenting with a single pre-training objective, the task-specific objective we devised (*i.e., Injected-Mutants Fixing* — IMF) is the best one, followed by MLM, RTD, and NSP. The overall best performing combination of objectives for each task is highlighted with yellow text (*e.g.,* IMF + MLM for bug-fixing). Given the high number of statistical tests performed (*i.e.,* each combination of pre-training objectives has been contrasted against all others, resulting in 45 tests per task), we provide full tables with the adjusted p-values and ORs in our replication package [repe].

Three main lessons can be learned from our results. First, the choice of the pre-training objective can make a substantial difference in the performance of transformers. Within each task, contrasting the best-performing combination with the worst-performing one results in statistically significant differences (p-value < 0.001) with ORs of 2.9 (*bug-fixing*), 38 (*code summarization*), 29 (*code completion*), 2.9 (*code & comment-to-code*), and 19.5 (*code-to-comment*). For example, in the case of *code summarization*, we move from the 4.73% of correct predictions ensured by MLM, down to the 1.94% achieved with the NSP objective.

Second, the 💡 high effectiveness of the MLM pre-training objective: MLM is involved, alone or in combination with other objectives, in all best configurations we found. Moreover, even in the *bug-fixing* and *code-to-comment* tasks in which the MLM objective in isolation is not the best-in-class, the difference in performance with respect to the best configuration (*i.e.,* IMF + MLM and NSP + MLM respectively) is not statistically significant.

Third, concerning the task-specific objectives we devised, we only observed a (not statistically significant) improvement over MLM for the *bug-fixing* task with the IMF objective. The latter provides the model with information substantially different from those that can be captured by MLM and it closely resembles the fine-tuning task. This might not be the case for the other task-specific objectives we devised. For example, the *Method Name Generation* MNG objective devised for *code summarization* is a sort of more specific version of MLM: rather than randomly masking 15% of tokens in the training instance, the method name is the only masked token the model has to predict. We conclude that 💡 task-specific pre-training objectives might boost performance if they (i) capture orthogonal information as compared to non-specific objectives such as MLM; and (ii) strictly simulate the downstream task. However, even in this case, the gain over the classic MLM objective may be limited and should be assessed empirically.

5.4 Conclusions

We investigated the impact on the performance of transformers [VSP+17] of the pre-training phase, nowadays adopted in most of the applications of these models to SE tasks. Two aspects have been investigated: (i) the extent to which pre-training helps the learning even when the task at hand allows to build very large datasets for fine-tuning; and (ii) the impact on the model's performance the choice of the pre-training objective(s) can have.

We found that when the size of the fine-tuning dataset is large enough, approaching that of the pre-training dataset, the pre-training phase is unlikely to help. Instead, it provides a substantial boost of performance for tasks in which the scarcity of training data leads to small fine-tuning datasets. We also observed the major role played by the choice of the pre-training objectives, with different combinations of objectives providing substantially different performance. Pre-training objectives specifically tailored for the downstream tasks can help but, at least in our study, did not result in a significant improvement of performance as compared to the classic *Masked Language Model* task.

These results also hold in the context of code review-related tasks, confirming our choice of the *Masked Language Model* as pre-training objective in Chapter 4 as a good one.

This chapter concludes our contributions on the usage of DL-based solutions for code review automation. Several subsequent works built on top of our techniques, proposing better solutions for automating the targeted tasks. Still, both our work as well as those proposed successively mostly focus on a quantitative evaluation of the proposed solutions. In the next chapter we look at the support these techniques are able to provide in terms of code review automation from a more qualitative point of view.

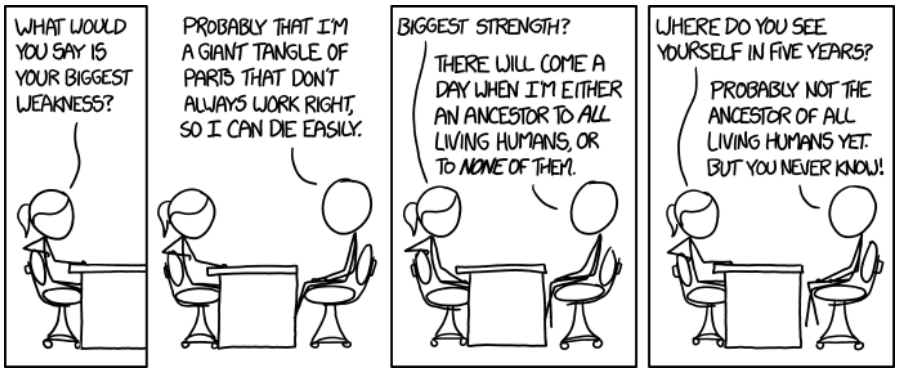
5.5 Replication Package

We release all code and data used in our study in a comprehensive replication package [repe, repf]. It contains:

- the material from the sistematic literature review (*e.g.,* used queries and collected papers);
- the processed and split datasets we used (both for pre-training and fine-tuning);
- the scripts used to preprocess and clean the data, as well as the necessary to generate mutants of given Java methods;
- the Google Colab notebooks we used to pre-train and fine-tune the models;
- the checkpoints of the best models obtained for each task, dataset and pre-training objective, as well as the trained tokenizer model and vocabulary;
- the statistical analysis, BLEU score and Levenstein distance of the models predictions;
- instructions to replicate our research.

6

Code Review Automation: Strengths and Weaknesses of the State-of-the-art



The code review automation techniques presented in this thesis ([TPT+21, TMM+22]) are only the first of several focusing on automating one or both of the *code & comment-to-code* and the *code-to-comment* tasks [TPT22b, LYJ+22, HTTA22, LLG+22]. These techniques have been evaluated on test sets containing hundreds of instances representative of the automated tasks. For example, for the *code & comment-to-code* task, the test sets feature $\langle C_s, R_{nl} \rangle$ pairs which are fed to the approach to assess whether it can address the reviewer's comment R_{nl} and generate the expected C_r. The outcome of these evaluations is a mostly quantitative report showing, *e.g.,* the percentage of instances in the test set for which the approach successfully generated a prediction.

However, such quantitative measures only tell part of the story. Indeed, it could happen that the approach is targeting the *low-hanging fruits*, being successful in only simple code review scenarios which are unlikely to save developers' time. For the *code & comment-to-code* task this might mean successfully addressing mostly comments requiring minor changes to C_s (*e.g.,* addition/removal of whitespaces to improve the formatting). Similarly, for the *code-to-comment* task the approach could overfit and mostly be successful in posting comments related to *e.g.,* replacing the `==` operator in Java with an `equals` invocation when needed.

In other words, little is known about the code review scenarios in which these techniques succeed or fail.

To fill this gap, we manually analyzed 2,296 predictions generated by three state-of-the-art techniques [TMM+22, HTTA22, LLG+22] (including the one presented in Chapter 4) automating the code review tasks previously described. The predictions have been generated on the original test sets used in the papers presenting the subject techniques.

The result of such an analysis are two taxonomies (one per task) featuring a total of 120 types of code changes requested during code review (*e.g., extract method refactoring*, *add thrown exception*) with indication about the extent to which state-of-the-art techniques are successful in (i) requesting their implementation when needed by properly commenting the submitted code as a human reviewer would do (*code-to-comment* task); (ii) automatically implementing them to address a reviewer's comment (*code & comment-to-code* task).

We found that the proposed techniques can provide support in a wide variety of code changes. However, there are areas of our taxonomies in which the approaches consistently fail, pointing to the need for more research. As a concrete example, the experimented techniques struggle when they need to recommend (*code-to-comment* task) or implement (*code & comment-to-code* task) complex code changes spanning across several code components. This is due to the "view" they have of the code base, usually limited to a single function or diff hunk submitted for review. This indicates the need for enriching the contextual information provided to these techniques.

During our manual analysis we also found that ∼25% of the instances in the inspected datasets are the result of data extraction errors possibly undermining the techniques' performance. We discuss the reasons for such problematic instances.

Finally, given the recent proposal of general purpose Large Language Models (LLMs) such as ChatGPT [cha], it is unclear what the actual need is for code review automation via specialized techniques. We compared the three subject approaches with ChatGPT, showing that, while the latter represents a competitive solution for the *code & comment-to-code* task, it suffers in the *code-to-comment* task.

The content of this chapter has been presented in the following paper:

> **Code Review Automation: Strengths and Weaknesses of the State of the Art**
> Rosalia Tufano, Ozren Dabić, Antonio Mastropaolo, Matteo Ciniselli, Gabriele Bavota. *Submitted to IEEE Transactions on Software Engineering (TSE) after Major Revisions*

6.1 Study Design

The *goal* of this study is to assess the capabilities of state-of-the-art techniques for code review automation. The *context* consists of: (i) three techniques, *i.e.,* Tufano R. *et al.* [TMM+22] (T5CR), Li Z. *et al.* [LLG+22] (CODEREVIEWER), and Hong *et al.* [HTTA22] (COMMENTFINDER); (ii) two code review tasks for which the subject techniques provide automation, *i.e., code-to-comment* and *code & comment-to-code*; and (iii) instances featured in the test datasets on which the techniques have been evaluated in the papers presenting them. We do not aim at comparing the performance of the three techniques to understand which one is the best. Rather, we look at them as a whole to understand the status of code review automation.

We address the following research questions (RQs):

RQ$_1$: *What are the characteristics of correct and wrong recommendations generated by techniques for code review automation?* We cluster the predictions generated by the experimented techniques into two sets representing instances for which the approaches generated a correct or a wrong prediction. Then, we quantitatively compare these two sets. For the *code-to-comment* task we compare the "complexity" of the comments to automatically generate (*i.e.,* those present in the ground truth). Similarly, for the *code & comment-to-code* task, we compare the "complexity" of the code changes to implement. Such an analysis will shed some light on the extent to which the state-of-the-art techniques overfit towards the low-hanging fruits of the datasets.

On top of this, we qualitatively analyze 2,296 predictions generated by the three approaches (equally distributed between correct and wrong predictions) to characterize the type of code change they were able to request (*code-to-comment* task) or to automatically implement (*code & comment-to-code* task). The objective is to understand the scenarios in which these techniques are successful *vs* those in which they tend to fail. For example, if the outcome reveals that for the *code & comment-to-code* task the techniques are mostly successful in implementing formatting changes, but tend to fail when dealing with more challenging code changes (*e.g.,* fixing a bug), this would question their usefulness.

RQ$_2$: *To what extent are the datasets used to train and test techniques for code review automation suitable for such a scope?* Through qualitative analysis we unveil the presence of problematic instances in the datasets used in the subject studies, calling for better dataset-cleaning pipelines.

RQ$_3$: *How do techniques for code review automation proposed in the literature compare to state-of-the-art large language models?* We compare the three subject techniques with ChatGPT [cha] as representative of LLMs. Such a comparison informs the need to further invest in automating code review through specialized models rather than relying on general-purpose LLMs.

6.1.1 Study Context

We present the study context in terms of experimented techniques and datasets/predictions we analyzed.

Techniques for Code Review Automation

Our work focuses on techniques aimed at *imitating* human reviewers, thus automating the *code-to-comment* and/or *code & comment-to-code* task. Based on our analysis of the literature, five approaches target these tasks: Tufano R. *et al.* [TPT+21], T5CR [TMM+22], COMMENTFINDER [HTTA22], CODEREVIEWER [LLG+22], and Li L. *et al.* [LYJ+22]. Tufano R. *et al.* [TPT+21] has been excluded since in their followup work [TMM+22] the authors showed the superiority of the newly presented technique (T5CR) as compared to their first attempt [TPT+21] in automating code review. Li L. *et al.* [LYJ+22], instead, has been excluded after inspecting the replication package shared by the authors: We rely on the authors' replication packages to download (and then inspect) the predictions generated by their model on the test set. The replication package for Li L. *et al.* [LYJ+22] featured 987 predictions for the 1,055 instances in the test set, casting doubts on the mapping between test instances and predictions. Also, 7 of these predictions had one or more "partial duplicate" in the training set, meaning that the training set featured the same code instance (*i.e.,* the same code for which the technique had to automatically generate "reviewer's comments") of an entry in the test set with a different target message. While this is not a problem in principle, this plays a role in our qualitative evaluation, where we analyze whether the comments generated by these techniques are semantically equivalent to the expected one (despite they might use a different wording). Having the same code instance in the training set allows the approach by Li L. *et al.* [LYJ+22] to "reuse" the reviewer's comment from the training set, thus generating something that, for sure, will be meaningful. For these reasons we discarded this technique from our study. This left us with the three techniques.

For the *code-to-comment* task, we consider predictions generated by all three approaches with: T5CR [TMM+22] being representative of DL-based techniques working at method-level granularity and not considering the code diff as an input; COMMENTFINDER [HTTA22] being representative of IR-based techniques also working at method-level granularity; and CODEREVIEWER [LLG+22] being representative of DL-based techniques working at "diff hunk" granularity and considering the code diff as an input. For the *code & comment-to-code* task we only consider T5CR and CODEREVIEWER, since COMMENTFINDER does not provide support for such a task.

Datasets and Predictions

From the replication packages of the three techniques [TMM+22, HTTA22, LLG+22] we collected the test sets used for their evaluation and the corresponding predictions. The size of the test datasets is reported in Table 6.1. There are two main differences among the datasets. The first is in the representation of the code submitted for review that is provided as input to the technique (C_s). While for T5CR [TMM+22] and COMMENTFINDER [HTTA22] C_s is a Java method, for CODEREVIEWER [LLG+22] is a diff hunk. The second concerns the fact that T5CR and COMMENTFINDER only work with Java code, while CODEREVIEWER supports nine languages, including Java. In our study we only considered Java instances for consistency and to simplify the following described manual analysis.

Table 6.1. Summary of related work. For Li Z. *et al.* [LLG+22] the size of the test set refers to Java instances only.

Reference	Task	Underlying technique	# Training instances	# Test instances
Tufano M. *et al.* [TPW+19]	*code-to-code*	NMT	8.6k	1k
Tufano R. *et al.* [TPT+21]	*code-to-code* *code & comment-to-code*	NMT	13.7k	1.7k
Tufano R. *et al.* [TMM+22]	*code-to-code* *code & comment-to-code* *code-to-comment*	T5 (pre-trained)	134.2k	16.7k
Thongtanunam *et al.* [TPT22b]	*code-to-code*	NMT	118k	14.7k
Li L. *et al.* [LYJ+22]	*code & comment-to-code* *code-to-comment*	T5 (pre-trained)	87k	1k
Hong *et al.* [HTTA22]	*code-to-comment*	IR	13.7k	1.7k
Li Z. *et al.* [LLG+22]	*code & comment-to-code* *code-to-comment* *code quality estimation*	T5 (pre-trained)	117.7k 150.4k 265.8k	2.2k 1.6k 66.4k

6.1.2 Data Collection and Analysis

RQ_1: Correct *vs* wrong recommendations

To answer RQ_1 the first step is to classify the predictions by the three approaches as correct or wrong. We considered a prediction as correct if it represents an exact match (EM) with the target (*i.e.,* the expected output). This means that for the *code-to-comment* task the model generated a comment identical to the one written by human reviewers, while for the *code & comment-to-code* task the model implemented a code change required by the reviewer exactly as the human contributor did. For each pair of technique and automated task, such a process resulted in the identification of the buckets of correct and wrong predictions reported in Table 6.2 (columns "# correct (%)" and "# wrong (%)"). While the EM metric has been used in the evaluation of the three techniques, we acknowledge that it has strong limitations, since it provides quite a strict definition of correctness. For example, an automatically generated natural language comment requesting the same code changes of the target comment with different wording is considered wrong. While this undermines a purely quantitative assessment of performance, in our study we use EM only as a mean to identify *candidate* correct and wrong predictions. The predictions will undergo a manual analysis which, for example, considers correct generated messages being semantically equivalent to those posted by the human reviewers.

Qualitative Analysis. The goal of the manual analysis was to characterize the type of code change the experimented techniques were (or were not) able to request (*code-to-comment* task) or to automatically implement (*code & comment-to-code* task). For each of the above-described buckets of correct and wrong predictions (as identified via the EM analysis), we targeted the manual inspection of 167 valid instances, corresponding to a statistically significant sample with at least a confidence level of 99% and confidence interval of

Table 6.2. Inspected instances

Reference	Task	# correct	# wrong	Inspected correct	Inspected wrong	Valid correct	Valid wrong
T5CR [TMM+22]	*code&comment-to-code*	2,363	14,417	178	272	199	167
	code-to-comment	354	16,426	200	227	169	189
COMMENTFINDER [HTTA22]	*code-to-comment*	479	16'301	234	254	169	176
CODEREVIEWER [LLG+22]	*code&comment-to-code*	599	1,607	198	321	197	176
	code-to-comment	0	1,611	-	412	50	179

±10% for each bucket. The target of 167 instances was defined by computing a sample size (SS) calculation formula [Ros11b] on the bucket having the largest "population" (*i.e.,* wrong predictions generated by T5CR for the *code-to-comment* task, with 16'426 instances):

$$SS = \frac{\frac{z^2 \times p(1-p)}{e^2}}{1 + (\frac{z^2 \times p(1-p)}{e^2 \cdot N})}$$

where p is the predicted probability of the observation event to occur, set to 0.5 when not known a priori (as in our case), N is the population size, e is the estimated margin of error (±10%), z is the z-score for a given confidence level (in our case, 2.58 for the 99% confidence). As it can be seen from the formula, the larger N, the larger the sample size. Thus, using the largest "population" to compute the number of instances to inspect is a conservative choice, ensuring even better confidence when working on smaller buckets.

We use the term "valid" instances to account for the following scenarios. First, when inspecting a prediction falling in one of the *wrong* buckets it is possible that we realize that the prediction is actually correct (*e.g.,* the comment generated/retrieved by the technique uses different wording as compared to the target, but it is semantically equivalent). In this case, while we inspected a *wrong* instance, it will actually fall into the corresponding *correct* bucket. Second, we discarded several problematic instances we found in the test sets of the experimented techniques. For example, we found instances in the *code & comment-to-code* task for which, given the input code as context, it was impossible even for a human to understand the associated reviewer's comment (*i.e.,* what the reviewer was asking to change in the input code). Indeed, the reviewer's comment referred to a wider context (*e.g.,* parts of the code base not provided as input to the model), making the prediction impossible for the automated technique. These are problematic instances in the test set rather than failure cases of the technique, and we document them in RQ_2. In summary, an instance was considered "valid" if, given the input information (*i.e.,* C_s for the *code-to-comment* task, and $\langle C_s, R_{nl} \rangle$ for the *code & comment-to-code* task), it was possible for a human to understand the rationale for the related output: For the *code-to-comment* task, this means that the human evaluator was able to understand what the R_{nl} comment to generate refers to (*i.e.,* what the problem in the submitted code C_s spotted by the reviewer is); for the *code & comment-to-code* task, the evaluator considers an instance valid if the changes resulting in the revised code C_r to generate actually address the reviewer comment R_{nl} posted for the submitted code C_s.

The instances to inspect were randomly selected from each bucket until the target number of valid instances was reached. The columns "Inspected correct" and "Inspected wrong" in Table 6.2 report, for each bucket, the number of instances we ended up manually inspecting to reach our target of 167 valid instances per bucket.

Overall, we inspected 2,296 instances. Each instance has been independently inspected by two of the researchers involved in this work (from now on, evaluators) who were tasked with classifying the type of change to request (*code-to-comment*) or to implement (*code & comment-to-code*). Five researchers were involved in the manual analysis. On average, they have 13.4 years of programming experience (min=6) and 9.2 years of experience with Java (min=5), the language used in the inspected datasets. One of them holds a PhD in software engineering, and three more are currently pursuing a PhD in software engineering. One is a software engineer.

The whole process was supported by a web app we developed that implemented the required logic and provided a handy interface to visualize the instance to label. For each instance, the evaluator was presented with: (i) the input provided to the approach; (ii) the generated prediction; and (iii) the expected output. As a result of the inspection, the evaluator could classify the instance or discard it as non-valid, providing an explanation as to why it was discarded.

The classification required to assign the instance one or more labels describing the change (*e.g., refactoring → extract method*). Each evaluator was free to define their own label(s) (*i.e.,* open coding procedure), as they felt it was needed to properly describe the change: For this specific task, it was not possible to define upfront all possible labels, making card sorting [Cox99] not suitable for our study. Indeed, while there are taxonomies of issues identified during the code review process [ML09, BBZJ14, PSP+18] their abstraction level is not suitable for our goal. For example, the taxonomy by Mäntylä *et al.* [ML09] includes a category named *evolvability defects → structure* which is too coarse grained to investigate the automation capabilities of the subject techniques.

To provide a concrete example, the taxonomy depicting the types of changes that the three techniques were (or were not) able to automatically request in the *code-to-comment* task (Fig. 6.1) we have an entire tree dedicated to the recommendation of refactoring operations (which would fall under the *evolvability defects → structure* taxonomy from [ML09]). Our taxonomy includes concrete refactoring operations (*e.g., Extract method*, *Change variable/constant type*), some of which are successfully recommended by the experimented techniques, while others consistently represent failing scenarios. The fine-grained nature of our taxonomy allows to observe these differences.

The agreement among the evaluators was to define each label in the form *parent → child*, where *parent* was a coarse-grained description of the change while *child* was a more specific, fine-grained description. New labels defined by an evaluator were made available in the web application to the other evaluators. While this goes against the notion of open coding, this allows to reduce the chance of multiple evaluators defining similar labels to describe the same type of change while not substantially biasing the process. The evaluator was also in charge of flagging instances in the wrong buckets as "actually correct" in case they felt that the prediction, while different from the target, was correct.

The manual evaluation was performed in three rounds. A first round asked each evaluator to inspect 30 instances. This round resulted in a set of labels that has been inspected by the evaluators with the goal of merging similar labels and come up with a common strategy to categorize the instances in the next rounds. Then, a second round was performed in which the evaluators targeted the labeling of 30% of the overall instances assigned to them. Again, such a round was followed by a further inspection of the defined labels, with grouping of similar labels and further discussion on strategies to improve agreement.

The rationale for the number of instances to inspect in each of the three rounds was the following. We wanted to label very few instances in the first round (30) since we expected several inconsistencies in the way in which the evaluators were going to perform the labeling and, thus, we targeted a short dry run to test the adopted web application, the clarity of the overall process and, at least in part, the type of labels assigned by the evaluators (*e.g.,* their granularity). Then, we decided to follow with a larger second batch (30%) which allowed to spot more corner cases worth to be discussed among the evaluators (*e.g.,* instances for which a researcher was unsure about the type of label to assign). Finally, since we felt that the labeling process was well-defined and clarified among the evaluators, we decided to move on labeling the whole dataset.

Once all 2,296 instances have been inspected by two evaluators, we solved conflicts that arose in 1,225 cases[1]. Such a number may look high, since it represents 53% of the inspected instances. However, the high rate of conflicts is explained by three design decisions. First, we considered all conflicts, also the ones resulting in the first and second round in which the set of possible labels was not stable at all. Second, the labels in our study emerged from the data and were not pre-defined. To get an idea of the complexity of this task the whole process resulted in a total of 120 different labels. Third, we were conservative in our definition of conflict, which occurred if: (i) two evaluators assigned a different set of labels to the instance, even if the two sets partially overlapped; (ii) two evaluators assigned the exact same set of labels to a *wrong* instance with only one of the two reporting the instance as "actually correct"; (iii) only one of the two evaluators labeled the instance, while the other one discarded it. Each conflict has been inspected by two additional evaluators, who discussed and solved it.

Finally, we used the assigned labels to build hierarchical taxonomies showing the types of changes in which the three techniques tend to succeed and fail for the two automated tasks. Such a process required additional inspections of the considered instances. Indeed, once all categories in the taxonomies have been defined, two of the evaluators rechecked that all instances were assigned to the most proper category. Indeed, it is possible that a category C added during the very last labeling round would be more suitable for instances inspected at the very beginning of the manual process, when C was not available (since no one came up with that label while inspecting the instance). This resulted in the re-assignment of 16 instances ($\sim$1%).

The final number of valid instances (*i.e.,* non-discarded) within each bucket is reported in the columns "Valid correct" and "Valid wrong" in Table 6.2.

[1]Given the open nature of the coding, it was not possible to compute a meaningful inter-rater agreement (*e.g.,* Cohen's kappa).

A few clarifications are needed for values which are different from 167, which was our original target. First, we did not have correct (EM) predictions generated by CODEREVIEWER [LLG+22] for the *code-to-comment* task. Thus, we applied the following procedure to collect instances for the corresponding bucket. We selected among the *wrong* predictions generated by CODEREVIEWER, the top-100 in terms of BLEU-4 (Bilingual Evaluation Understudy) score [PRWZ02]. BLEU measures how similar the candidate (predicted) and reference (oracle) comments are in terms of overlapping 4-grams. A value of 1.0 indicates that the candidate and the predicted comment are identical. The selected top-100 predictions have a BLEU-4 ranging between 0.28 and 0.72.

Our assumption is that *wrong* predictions having a high BLEU score are likely to be correct, since they closely resemble the target comment. During the manual analysis, we discarded the instances that despite the high BLEU score, were actually wrong, since they did not belong to the "correct bucket". This process led us to 50 valid instances in this bucket, which is the only one being underrepresented (see Table 6.2). Second, as it can be seen from Table 6.2, in several buckets we collected more than the targeted 167 valid instances. This is due to the conflict resolution phase in which some instances discarded by one of the two evaluators were considered valid and re-introduced.

The output of this analysis are two taxonomies reporting, for each of the two tasks, the types of code changes on which the experimented techniques tend to succeed or to fail.

Quantitative Analysis. We contrast the complexity of the test set instances resulting in correct and wrong predictions of the experimented techniques.

For the *code & comment-to-code* task, we measure the number of AST-level changes required to convert C_s (*i.e.,* the code submitted for review) into C_r (*i.e.,* the revised code addressing the reviewer's comment). We expect a higher number of changes to indicate a higher complexity of the comment to implement. The AST-level changes have been extracted using Gumtree Spoon AST Diff [FMB+14].

For the *code-to-comment* task, we used as proxy for complexity (i) the number of words featured in the comment to generate, under the assumption that longer comments are likely more complex, and (ii) the number of AST-level changes required to address the reviewer's comment (as done for the *code & comment-to-code* task). The latter was only computed for the predictions generated by T5CR and by COMMENTFINDER, since for CODEREVIEWER we did not manage to retrieve from the dataset the code implementing the required change, but only the submitted code with the posted reviewer's comment.

For both tasks, we report boxplots of the distribution of the complexity proxies for correct and wrong predictions both overall and by approach. We also statistically compare the two distributions assuming a significance level of 95% and using the Wilcoxon test [Wil45]. The Cliff's Delta (d) is used as effect size [GK05a], 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 $|d| \geq 0.474$ [GK05a]. We adjust p-values using Holm's correction procedure [Hol79a]. We compare the complexity proxies only on the predictions we manually validated. The reason is that, as explained, relying on EM to identify correct predictions could lead to false negatives, thus invalidating our quantitative analysis.

RQ_2: Datasets quality

The datasets used to train and test the experimented techniques have been automatically mined from GitHub. The authors applied a number of heuristics to filter-out problematic instances. For example, in the *code-to-comment* task efforts have been made to remove review comments posted by bots. Similarly, in the *code & comment-to-code* task in which $\langle C_s, R_{nl} \rangle \rightarrow C_r$ triplets are involved, checks are performed to make sure that C_r (the code which should implement the reviewer's comment R_{nl}) is different from C_s. Indeed, $C_s = C_r \implies R_{nl}$ not addressed.

Despite the effort in cleaning the datasets, in our manual analysis we found ∼25% of the inspected instances representing noise in the datasets. As previously explained, when discarding an instance during the manual analysis the evaluators had to report the reason why said instance was discarded. After such a process, the five evaluators looked at the provided motivations and distilled them into four main categories representing errors introduced during the automated mining of the data from GitHub. We answer RQ_2 by presenting statistics summarizing such an analysis.

RQ_3: Comparison with LLMs

We assess the performance of ChatGPT [cha] on the two tasks focus of our study. We limited the number of samples to 250 for ChatGPT, due to the high cost of running such an evaluation. Indeed, there were two manual steps to perform. First, we needed to interact with the ChatGPT GUI to manually prompt each instance on which we wanted to run ChatGPT. Based on some tests we performed, we ended up selecting the following two prompts for our tasks:

code-to-comment: `Write a code review of the following code "{inputCode}".`

code & comment-to-code: `Revise this code "{inputCode}" given this comment "{inputComment}".`

In the prompts `{inputCode}` and `{inputComment}` represent the C_s and R_{nl}, respectively, in the test datasets used for the two tasks. The replies provided by ChatGPT when using these prompts made it clear that it properly interpreted the task to perform.

Second, once ChatGPT generates an answer, we cannot rely on EM to check if it is correct, since ChatGPT has not been trained to generate answers in the same format used in the test sets. For this reason we had to manually inspect the generated answers to assess their correctness. Each generated answer was independently inspected by two of the involved researchers, who classified it as correct or wrong. For the *code-to-comment* task we considered the code review generated by ChatGPT as correct if it contains the target comment. For *code & comment-to-code*, we verified whether ChatGPT properly addressed the reviewer's comment, even with the coding solution being different to the target one. Conflicts (*i.e.,* the two researchers disagreed on the correctness of ChatGPT's answer), which arose in 18% of cases, were solved by a third researcher. For the *code-to-comment* task we randomly selected 50 instances from the test set of each of the experimented techniques (150 instances in total). The 50 instances included 25 correct (*i.e.,* the corresponding technique generated a correct solution) and 25 wrong predictions.

The same approach has been used for the *code & comment-to-code* task which, however, is only automated by two of the three subject techniques, thus resulting in 100 randomly selected instances.

We answer RQ_3 by reporting the percentage of cases in which ChatGPT was successful in both tasks. We also analyze the overlap between the state-of-the-art techniques and ChatGPT by reporting the percentage of cases in which (i) both succeed; (ii) at least one of the two succeeds; and (iii) none of the two succeeds.

6.2 Results Discussion

We discuss the achieved results by RQ. We use the 💡 icon to mark lessons learned and directions for future work.

6.2.1 RQ_1: Correct *vs* wrong recommendations

Fig. 6.1 and Fig. 6.2 report the taxonomies of "types of code changes" involved in the predictions generated by the subject techniques. Fig. 6.1 refers to the *code-to-comment* task, depicting types of changes that the techniques were supposed to ask for in generated comments, as a human reviewer would do. Fig. 6.2 refers to the *code & comment-to-code* task, reporting types of changes that the techniques were required to automatically implement.

The taxonomies include several trees, each one representing a generic set of code changes specified in the root category (*e.g., refactoring*, *bug-fix*). The number on the top-right corner of each label reports the number of instances we manually assigned to that change type (*e.g.,* 503 refer to *refactoring* in Fig. 6.1).

Three clarifications are needed on this point. First, one instance we analyzed (*i.e.,* a prediction generated by an approach for a test entry) could have been assigned to multiple categories since requiring multiple types of changes. Second, for readability reasons, we decided to not report in the picture all categories of code changes that have been assigned to less than ten instances. Indeed, it is difficult to draw any conclusion with so few data points. The full data is available in our replication package [repa].

Third, being a hierarchical taxonomy, one may expect the number of elements in a parent category to match the sum of the number of elements in its child categories. However, this is not the case due to two reasons. First, sometimes the parent category has been used as a label when we did not manage to clearly identify the required code change, but only its overall goal (*e.g.,* using *bug fix* → *fix wrong behavior* instead of its child *modify if condition*). Second, as previously explained, we do not depict in the taxonomies categories featuring less than 10 instances. However, we still count their instances in the parent category (*e.g., refactoring* → *renaming* has a *rename class* child which has not been depicted but contributes with 6 instances).

In this scenario, the parent category has been used as a label when we did not manage to clearly identify the required code change, but only its overall goal (*i.e.,* fixing wrong behavior). This is another reason why parent categories can have a higher counting than the sum of their child categories.

Figure 6.1. Taxonomy of types of changes for the *code-to-comment* task. The color assigned to each label reflects the ability of the techniques to automate the code review task in the context of such a change type (white best, black worst). We report the percentage of successful predictions by each approach for each change type as bars below the corresponding category: T5CR (blue bar), CODEREVIEWER (green), and COMMENTFINDER (red).

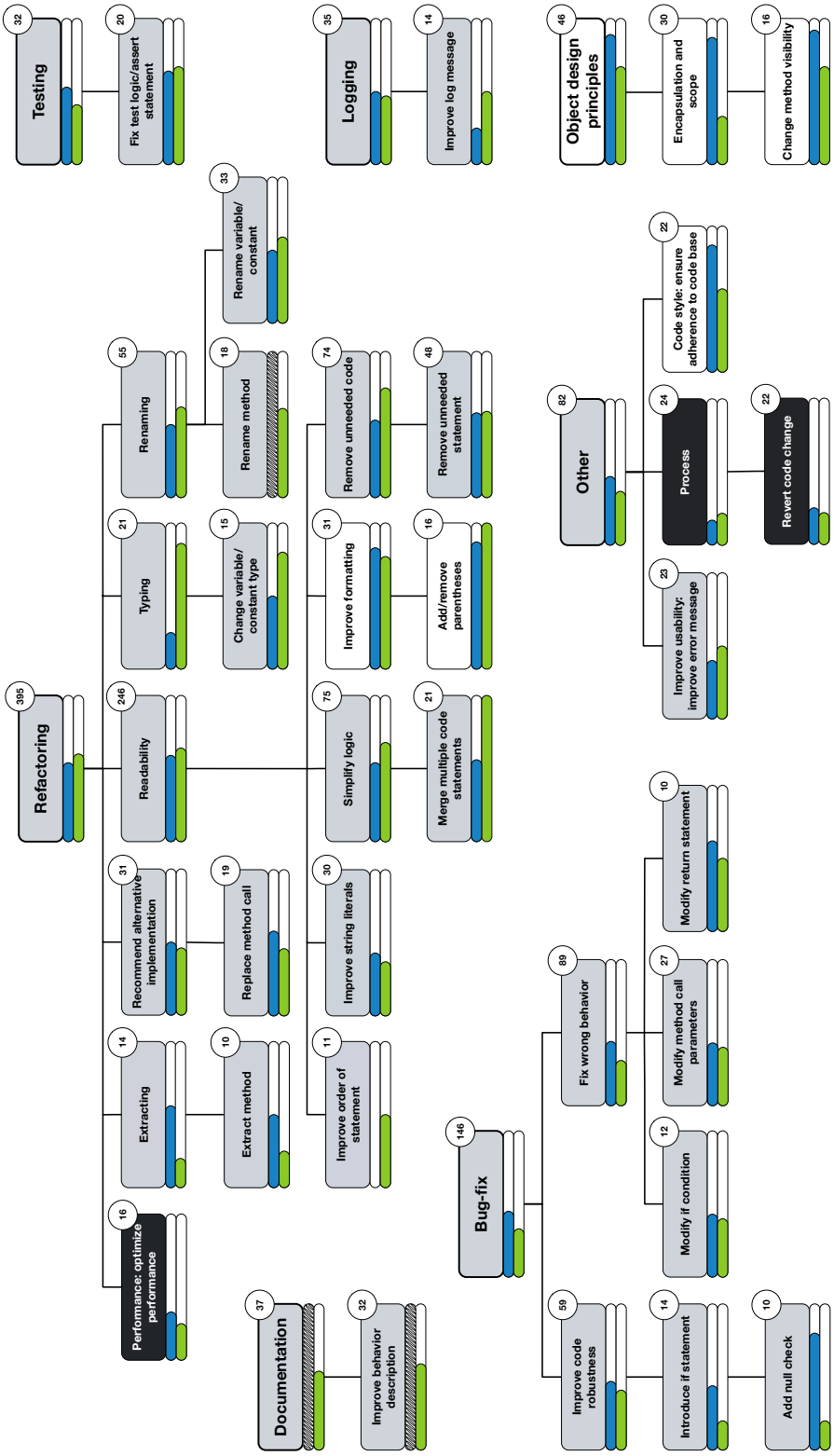


Figure 6.2. Taxonomy of types of changes for the *code & comment-to-code* task. The color assigned to each label reflects the ability of the techniques to automate the code review task in the context of such a change type (white best, black worst). We report the percentage of successful predictions by each approach for each change type as bars below the corresponding category: T5CR (blue bar), CODEREVIEWER (green).

The color assigned to each label reflects the ability of the techniques to automate the code review task in the context of such a change type. Since we manually analyzed ∼50% of correct and ∼50% of wrong predictions generated by each approach, a success rate around 50% for a change type indicates that the techniques do not tend to perform particularly well or bad for that change type.

Indeed, the goal of this analysis is to see if the correct (wrong) predictions are polarized by specific categories. For these reasons, we defined the color schema as follows:

A gray label indicates a change type for which the automation level aligns with what expected based on our sample of correct and wrong predictions. Looking at Table 6.2 it can be seen that, for the *code-to-comment* task, we inspected a total of 388 correct and 544 wrong instances. Such an imbalance is due, as previously explained, to the CODEREVIEWER technique which generated 0 EMs and for which we decided to manually inspect 100 wrong predictions looking for actually correct ones (50 identified). This means that we should expect an average performance per change type close to (388*100)/(544+388)=42%. For this reason, Fig. 6.1 features a grey category if the techniques succeeded for such a change type in 32% to 52% of cases (*i.e.,* 42% ± 10%). With a similar computation, Fig. 6.2 features a grey category if the techniques succeed in 43% to 63% of cases (since an average of 53% correct predictions has been analyzed per approach). Note that the "± 10% choice" has been done to simplify the results discussion and visualization. We acknowledge that other choices are possible (*e.g.,* ± 20%); raw data with exact percentages are available in our replication package [repa].

A black label indicates a change type for which the automation tends to fail, thus in which the techniques are struggling. This means a success rate lower than 32% for the *code-to-comment* task, and lower than 43% for the *code & comment-to-code* task.

A white label indicates a change type for which the automation succeeds more than expected, namely in at least 53% of cases for *code-to-comment* and 64% of cases for *code & comment-to-code*.

While this 3-level color schema represents the capabilities of the experimented techniques as a whole, Fig. 6.1 and Fig. 6.2 also report the percentage of successful predictions generated by each approach for each change type as bars below the corresponding category. In Fig. 6.1 the three bars are ordered from top to bottom as: T5CR (blue bar), CODEREVIEWER (green), and COMMENTFINDER (red). In Fig. 6.2 the bars are only two, corresponding to T5CR (blue) and CODEREVIEWER (green). An empty bar indicates that the approach always failed for instances of that type. Instead, the filling of the bar with a zig-zag pattern indicates that the manually inspected test set entries on which the corresponding technique has been experimented did not contain any instance of that type.

On top of the two taxonomies, Fig. 6.3 depicts the boxplots showing the computed "complexity" of the test instances on which the experimented techniques succeed (blue) or fail (orange). We discuss our qualitative and quantitative results by automated task.

Table 6.3. Non-EM classified as correct during manual analysis

Reference	Task	% correct non-EM
T5CR	*code & comment-to-code*	15.66%
	code-to-comment	2.58%
COMMENTFINDER	*code-to-comment*	2.22%
CODEREVIEWER	*code & comment-to-code*	17.37%
	code-to-comment	21.83%

Code-to-comment

Before looking at characteristics of correct and wrong predictions, Table 6.3 reports, for each approach and for each task, the percentage of non-EM predictions that we classified as actually correct. As it can be seen, 21.83% of non-EM predictions generated by CODEREVIEWER are actually correct for the *code-to-comment* task. The percentage is smaller for the other two techniques for which we did not focus the inspection on predictions having a high BLEU, but still non-negligible (∼2.5%). For example, in the case of COMMENTFINDER the EM predictions are 2.85% of the test set instances, while we found an additional 2.22% of non-EM predictions which are actually correct, almost doubling the approach's correctness. A manual analysis of (a sample of) non-EM predictions is needed to better assess the capabilities of an automated technique. An example of non-EM generated by CODEREVIEWER and being actually correct belongs to the *other* → *reuse existing code* category: The target comment was "Use `IOUtils` instead", while CODEREVIEWER generated the equivalent "Can we use Guava's `IOUtils` here?". This is a first important outcome of our study: 💡 Using EM to assess the automation of the *code-to-comment* task might be unfair.

Not surprisingly the white categories (*i.e.,* the techniques tend to succeed) are characterized by simple requests to include in the generated message, and in particular the *removal/-addition of a thrown exception*, and the *replacement of an operator*. The excellent performance achieved in these change categories are usually driven by the success of the COMMENTFINDER IR-based technique. The latter has 100% accuracy in recommending the addition/removal of exceptions, thanks to the retrieval from the training set of past reviewers' comments requiring such a change for similar methods. 💡 Looking at the taxonomy, it is clear that for code change types which are quite general, simple, and thus likely to be requested over and over again in different code review instances (*e.g.,* the addition/removal of exceptions, asking to *revert a code change*), an IR-based approach can be a trump card.

Differently, comments requiring the description of more complex changes are challenging to retrieve or synthesize. The *refactoring* tree provides interesting examples. Simple refactorings such as *changing variable/constant type* or *renaming variable/constant* are overall well-supported (*e.g.,* COMMENTFINDER: "`qry` $->$ `query`"). When it comes to refactorings involving complex code changes, possibly impacting multiple code components, the techniques tend to fail. This is the case for refactorings *extracting* or *moving* code elements. This is likely due to the limited contextual information provided to these techniques.

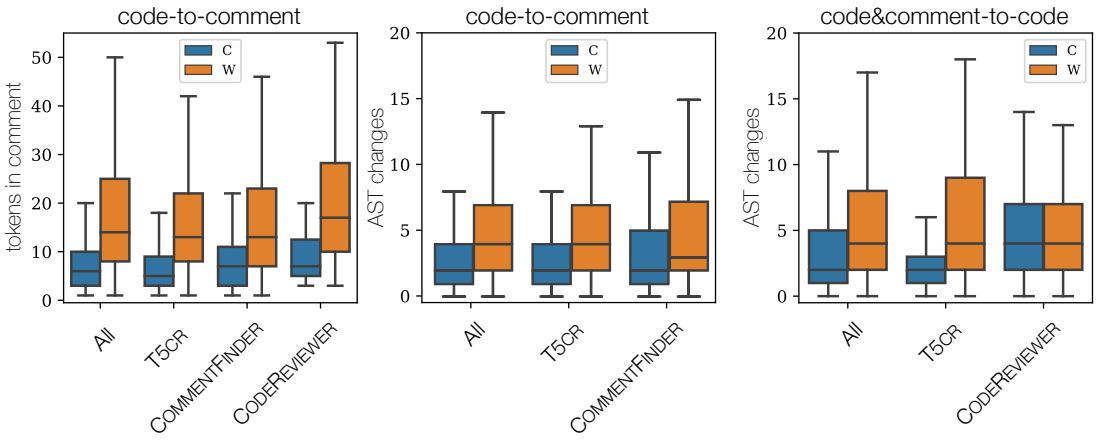


Figure 6.3. Task complexity for correct and wrong predictions

Among the experimented techniques, T5CR and COMMENTFINDER work at method-level granularity, meaning that the method provided as input represents everything the model knows about the system. Similarly, the "view" of CODEREVIEWER is limited to a diff hunk. Such an issue does also affect the performance of the techniques for apparently simple changes to require, such as those asking to *change the method visibility*. Indeed, without additional knowledge of the system it is difficult to judge what the correct visibility of a method should be. 💡 Pushing the boundaries of code review automation for these types of changes requires enriching the contextual information provided to the techniques, similarly to what observed for other software engineering tasks [TT22].

A negative exception in the refactoring tree is the *renaming of methods* which one could expect to be on a similar level of difficulty as compared to the *renaming of variable/constants* which is, instead, quite successful. We inspected these instances and we noticed that, while renaming variables/constants may just require a term expansion (as in the `qry` example previously reported), methods' names are more expressive and challenging and, while the techniques are sometimes able to capture the need for a renaming, they fail in recommending meaningful alternatives.

The techniques also have a hard time automating logging activities, especially when it comes to suggest the *introduction of a log statement* or the need to *change the log level* (*e.g.,* from `error` to `warning`). Interestingly, for both method renaming and recommendation of log-related changes, specialized DL-based techniques have been proposed in the literature (see *e.g.,* Alon *et al.* [AZLY19] for renaming, and Mastropaolo *et al.* [MPB22] for logging). Based on the reported empirical evaluations, those techniques proven to be quite effective in these tasks. For example, Mastropaolo *et al.* [MPB22] presented LANCE, an approach able to correctly recommend fixes to the level of log statements in 66% of cases. 💡 While the code review automation techniques proposed in the literature target the automation of generic code changes, the adoption of specialized techniques might be more effective for specific change types. However, this might not be straightforward to do in the context of the *code-to-comment* task. Indeed, assuming the will to specialize a model for "commenting" on

a specific type of issue, the first needed ingredient is a training dataset, which might not be easy to collect. One may think to cluster reviewers' comments via lexical analysis and train a specialized model on each of those clusters. Nevertheless, a trade-off between cohesiveness of the clusters and availability of training data soon becomes evident: very cohesive clusters will result in highly specialized models which, however, are likely to benefit from very little training data (*e.g.,* only a few instances in which a reviewer's comment is suggesting to introduce a log statement). Larger clusters featuring more training data are instead unlikely to specialize the model for specific types of recommendation, thus again pushing it towards a generic recommender. A more promising approach might be to manually define "commenting patterns" for a specific type of change (*i.e.,* a standard sentence expressing the need for improving a certain aspect of the code, such as introducing a log statement). In this case, the training dataset could be built by parsing code changes performed during the change history of a project (*e.g.,* commits introducing a new log statement), independently from the availability of code review information for these changes. This implies the possibility to reliably identifying code changes in which the target issue has been fixed. For some of the "black categories" in our taxonomy this can be easily achieved (*e.g.,* lack of log statement, need for changing the method visibility, etc.). Others would require more advanced solutions, like the usage of tools to detect refactoring operations [TKD22]. Negative instances, *i.e.,* code components on which the target issue does not manifest (*e.g.,* no need to add log statements), might be needed as well. Once trained, specialized models can be triggered on the code change submitted for review, reporting the improvement recommendations (if any) to the developer.

Not surprisingly, the experimented techniques do not shine in recommending types of code changes which are likely to be system-specific and, thus, difficult to learn/retrieve from other sources. This is the case for the *performance optimizations* comments that the techniques were required to emulate (*e.g.,* TARGET: "We should use `keyService` here, intention is to cache key temporary so under heavy load we don't download keys all the time"). 💡 A possible strategy to overcome this limitation could be to fine-tune the techniques to a specific software project or, at least, to a set of projects falling in the same domain (*e.g.,* DB engines). For example, after a pre-training performed on generic code review data, a DL-based approach could be fine-tuned to specifically support code review in a project. A major obstacle is the possible lack of fine-tuning training data, since a single project is unlikely to provide enough training instances. This may be partially overcome through data augmentation techniques [YWW22].

T5CR and COMMENTFINDER achieve good performance when it comes to asking for *testing*-related changes, correctly generating comments aimed at both improving the test coverage/logic (*e.g.,* T5CR: "add a check here to verify that the `serialDataReceived method` was not called") and cleaning/refining them, *e.g.,* T5CR suggested to replace an empty `String` passed as parameter in an assert statement with an `EMPTY_VALUE` constant already used in other statements of the test. In general, the changes to recommend in the *testing* category are very specific and tend to focus on a single code statement. 💡 Still, this shows the potential of these techniques as possible "refinement tools" for approaches supporting the automated generation of test cases [PE07, FA11].

From the quantitative perspective, the boxplots in Fig. 6.3 for the *code-to-comment* task suggest, as expected, that the techniques tend to succeed in the generation of simple reviewers' comments, having a median of 6 words composing them. As a comparison, the failing cases are more than twice longer (in terms of median), with 14 words. Such a trend holds, with minor differences, for all approaches. Also, it is interesting to notice that, when considering all techniques together, the first quartile of the wrong predictions is very close to the third quartile of the correct predictions, indicating a strong difference between the two sets that is confirmed by the statistical analysis with a p-value<0.001 accompanied by a large effect size (test results in [repa]). Similar observations can be drawn when using the AST changes required to implement the reviewer's comment as a complexity proxy. 💡 Future work should focus on boosting performance in these challenging scenarios, since the approaches seems to work pretty well in the generation of simple comments (<20% of wrong predictions have less than 6 words).

Code & comment-to-code

Also for this task, we start by observing the substantial percentage of non-EM predictions which are actually correct — 15.66% for T5CR and 17.37% for CODEREVIEWER. 💡 This reinforces the need for manual analysis when assessing the performance of techniques for code review automation.

The taxonomy depicted in Fig. 6.2 is smaller as compared to the previous one, since a higher number of categories (91) count less than 10 instances (for the full taxonomy see [repa]). It is interesting to see some major differences as compared to the previous taxonomy. Changes which were trivial to ask for in a comment to generate (*e.g.,* "please revert this change") are challenging to automatically implement, as required in the *code & comment-to-code* task. Indeed, the reverting may require several code changes which are not necessarily easy to predict, especially if the full code diff is not part of the information available to the model.

Interesting is the complementarity between the two techniques that support this task. T5CR is very effective in changes related to *object design principles*, *e.g.,* handling issues related to *encapsulation and scope* of variables/methods, which usually require minor code changes. Also, T5CR works well in implementing changes *ensuring adherence to the code base* in terms of *coding style*, *e.g.,* addressing comments like "`String.isEmpty()` is used in other places" pointing to an inappropriate `if` condition checking `coverageId.length() == 0`. CODEREVIEWER, instead, is the only approach supporting documentation changes, and can achieve excellent performance even for less-trivial code changes requiring *e.g.,* to *merge multiple code statements* (in order to improve readability) or to migrate towards more appropriate types (*refactoring → typing*). Considering that both techniques are built on top of a transformer-based architecture, such a complementarity can be partially explained by the different code representation they use, one looking at a single method at a time (T5CR) and one taking a diff hunk into account (CODEREVIEWER), possibly with a partial view of specific code components (*e.g.,* only the changed lines of a method are visible in the diff hunk). 💡 A hybrid representation including both the full representation of the involved code entities and the

changed lines might help in getting the best of both worlds.

As expected, both approaches are effective in the automation of very simple changes related to *improve the formatting* of code (*e.g., add/remove parentheses*, *add/remove white spaces*). The automation of these changes can be easily performed by a code formatter (*e.g.,* [pre]), without the need for expensive DL models. 💡 Such instances should be removed from the test sets used in the evaluation of techniques automating the *code & comment-to-code* task, to avoid inflating the percentage of EM predictions they generate.

The two techniques struggle to automatically implement complex *bug-fixes* requiring major code changes (*e.g., fix wrong behavior*) and/or changes to the code logic (*e.g., modify if condition*). 💡 This result is inline with what observed for techniques specialized in automated bug-fixing [TWB+19], which also tend to be successful in a minority of cases (usually <30%) confirming the need for more work in the area.

Finally, we want to comment on the performance of the two techniques on the 87 types of code changes which are not represented in Figure 2, since counting less than 10 instances each. Overall these 87 categories feature 132 of the predictions we inspected for the *code & comment-to-code* task.

Out of those, 69 (52%) were correct predictions, which matches the expected success rate and seems to suggest that the two state-of-the-art techniques do not really struggle in automatically implementing code changes which are likely to be less represented in the training set. However, by inspecting the predictions in these categories, we found out that a "good" level of performance (*i.e.,* ≥52% correct predictions) is only obtained for 48% of these poorly represented categories (*e.g.,* for 18 of them we observed 0% of correct predictions). It is thus possible that, in some cases, the models learn from similar and related categories in a sort of transfer learning fashion. For example, training on instances of the well-represented category "*remove unneeded statement*" might have played a role in achieving good performance on the five instances belonging to the code change type "*remove final modifier*". However, given the low number of instances in each of these categories (at most 9), we cannot draw any conclusion based on these findings.

The results of the quantitative analysis (right side of Fig. 6.3) show an interesting trend: while the correct predictions by T5CR require substantially simpler changes as compared to the wrong predictions (median of 2 AST changes *vs* 4, p-value < 0.001 with medium effect size), this is not the case for CODEREVIEWER. Here the two sets are basically equivalent in terms of code change complexity (negligible effect size). Looking at the boxplots it is clear that T5CR tends to overfit to simpler changes, while CODEREVIEWER, possibly thank to the diff hunk representation, is able to cope with more complex code changes as well, supporting its status as state-of-the-art approach.

6.2.2 RQ_2: Datasets quality

Table 6.4 reports the number of instances that we discarded as being problematic together with the reason why they have been discarded. For the sake of space, we shortened the *code & comment-to-code* task as C&NL2C and the *code-to-comment* as C2NL. While Table 6.4 shows also the results by test set of each technique, we focus the discussion on the overall trend (#). The "Other" category contains 16 instances discarded for various but rare reasons, that we do not discuss but document in [repa]. The remaining "discarding reasons" are sorted based on their frequency from top to bottom.

Table 6.4. Categories of discarded instances

Reason	#	T5CR		CommentFinder	CodeReviewer	
		C&NL2C	C2NL	C2NL	C&NL2C	C2NL
Unclear comment	284	32	55	97	60	40
No change asked	182	24	13	44	30	71
Ignored comment	74	25	0	0	49	0
Wrong linking	18	1	0	1	6	10
Other	16	2	1	1	1	11

For 284 cases, we assigned the *unclear comment* label to discard the instance since it was impossible even for a human to understand what to implement (*code & comment-to-code*) or what the target comment to generate was actually asking to the developer (*code-to-comment*). For the *code & comment-to-code* task, this was due to the limited contextual information provided to the model (*i.e.,* the input), meaning that the reviewer's comment referred to a larger code context not available to the model. For the *code-to-comment* task, a recurring problem is again the lack of context but, this time, related to the conversation that happened between the contributor and the reviewer(s), which is not visible to the techniques. For example, a comment saying "ah, ok, that would be clearer" is meaningless without knowing the previous exchanged messages. As already observed in RQ_1, increasing the contextual information is a must to push the boundaries of code review automation.

In 182 instances we inspected the reviewer's comment was not requesting any change. For the *code & comment-to-code* task, this means that the approaches could not really address the comment by modifying the code (*e.g.,* "Awesome work so far, Eli!"). For the *code-to-comment* task this means training and testing the technique for the generation of comments which are uninteresting given the automation goal. Indeed, these techniques aim at generating comments asking for code changes targeting the improvement of source code. Thus, comments like "I am not sure what GitHub wants to tell me with this icon here :)" should not be considered relevant for these approaches. The cleaning pipelines employed in the works presenting the experimented techniques fail in filtering out those meaningless instances.

In 74 cases (all related to the *code & comment-to-code* task), while the reviewer's comment was asking for a change, the contributor was changing the code but not to address the comment. These instances penalize both the learning and the evaluation of the models. Indeed, even assuming that the model correctly implements the change required by the

reviewer, during training the weights of the network will be revised to steer the prediction towards a different (wrong) target and, during evaluation, any quantitative metric is likely to point to a wrong prediction.

Finally, 18 instances result from errors while mining the dataset, since the code has been linked to a wrong code, *e.g.,* "there's no need for `final` in interfaces" for an input code not being an interface.

💡 Worth noticing is the overall number of discarded instances (574) out of the 2,296 manually analyzed (25%). Since the test set is just a random selection of 10% of data, we can assume a similar distribution in the training sets of the subject techniques. Thus, all discussed instances have the potential to hinder the training and bias the testing, since it is unreasonable to expect them to result in a correct prediction given the target. Supervised or unsupervised techniques aimed at removing these problematic instances are needed to make a further step ahead on code review automation thanks to higher-quality datasets.

6.2.3 RQ_3: State-of-the-art *vs* ChatGPT

Table 6.5 shows the results of the manual analysis assessing ChatGPT in automating code review: For each task (rows) we report the percentage of cases in which ChatGPT succeeds (✓) or fail (✗) for instances on which the state-of-the-art (SOTA) techniques were correct or wrong. Results are aggregated for the three approaches, with raw data available in our replication package [repa].

Table 6.5. Manual analysis of ChatGPT predictions

Task	✓State-of-the-art		✗State-of-the-art	
	✓ChatGPT	✗ChatGPT	✓ChatGPT	✗ChatGPT
code & comment-to-code	66%	34%	44%	56%
code-to-comment	19%	81%	7%	93%

ChatGPT performs slightly better than the SOTA in the *code & comment-to-code* task, being able to address the reviewer's comment in 83/150=55% of cases, as compared to the 50% of the three techniques (we selected half instances on which they work, and half on which they fail — see Section 6.1.2). Interesting is the complementarity between ChatGPT and the SOTA: ChatGPT succeeds in 44% of cases in which the SOTA techniques fail. These are mostly instances in which the reviewer's comment provides little information about the change to implement. For example, ChatGPT addressed a reviewer's comment pointing to a "*wrong formula*" in the code ("wrong formula" is the full content of the reviewer's comment) by applying the following change: `forward * strikesLike.get(i) + shiftOutput;` → `forward * Math.exp(strikesLike.get(i)) + shiftOutput;`. This is a failing instance for the SOTA.

Concerning the *code-to-comment* task, ChatGPT succeeds in 26/100=26% of cases, being less performant than the SOTA. Only 7 of those instances are failure cases for the SOTA. For this task the output of ChatGPT is not a single comment (as for the SOTA techniques) but a list of observations regarding the submitted code. In most of cases, we found these comments

to be meaningful, but they often miss or are in contrast with the point raised by the human reviewer.

This is the case for an instance in which the diff hunk reported a change from `trace` to `info` for the level of a logging statement. While the reviewer complained about this change (and the SOTA technique agreed with the human reviewer), ChatGPT was in favor of it, commenting: "an info level logging statement would be more applicable").

ChatGPT is a competitive baseline for the *code & comment-to-code* task. This might be due to the fact that it saw the code addressing the reviewer's comment during training. Thus, the extent to which such a comparison is fair is questionable. When it comes to generating reviewers' comments, SOTA techniques are superior, supporting the worth of further research in this direction.

6.3 Conclusion

We assessed the capabilities of three state-of-the-art techniques for code review automation [TMM+22, HTTA22, LLG+22]. Differently from the mostly quantitative evaluations available in the literature our study has a strong qualitative focus. Our study disclosed the scenarios in which state-of-the-art approaches tend to succeed and fail (RQ_1) and identified issues in the quality of the datasets used for their training and evaluation (RQ_2). Finally, we showed that ChatGPT [cha], as representative of Large Language Models, is a competitive technique for code review automation, but still struggles in several scenarios, justifying the need for more research on models specialized for such automation (RQ_3). The main outcome of this work is a research agenda for future work we will discuss in our concluding Chapter 7.

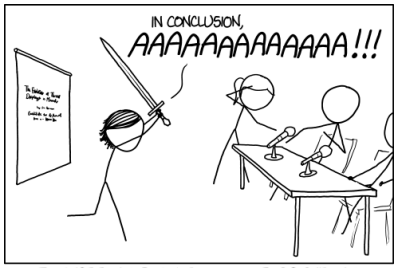
6.4 Replication Package

We release all data used in our study in a comprehensive replication package [repa]. It contains:

- all the instances inspected during the manual analysis (both valid and discarded ones);
- the instances used to evaluate the performance of ChatGPT-3 for both tasks and its predictions;
- the results of the statistical analysis on the instances complexity;
- all the necessary to compute the code generation complexity based on AST-changes between the input code and the code to generate in order to implement the reviewer comment;
- all the information, such as number of total istances, number of correct/wrong predictions, percentage of correct predictions, for each label used for both tasks.

7

Conclusions and Future Work



In this thesis we presented our research towards the automation of code review activities. We defined three tasks, *i.e., code-to-code*, *code & comment-to-code*, *code-to-comment*, and proposed to automate them by leveraging DL models and the large amount of code review data available from online platforms hosting open source projects (Chapters 3 and 4). Several research groups built on top of these techniques presenting better code review automation tools aimed at overcoming some of the limitations of our approaches [TPT22b, LYJ+22, HTTA22, LLG+22].

We also investigated the extent to which different pre-training objectives can help in maximizing the performance of DL models when applied to code related tasks, including the code review ones target of this thesis (Chapter 5).

Finally, we performed a deep qualitative assessment of the predictions generated by state-of-the-art techniques for code review automation (Chapter 6), including the one we presented in Chapters 4.

In the following we discuss the limitations that still affect code review automation techniques and future research directions.

7.1 Limitations and Future Work

7.1.1 A Community Effort for Large-scale and High-quality Code Review Datasets

Despite the best efforts invested in removing from the training/test datasets problematic instances (*e.g.,* reviewers' comments that do suggest any change to implement in the code) through pre-processing pipelines, we show in Section 6.2.2 that ∼25% of instances in code review datasets used in the literature (including ours) might be problematic. Higher quality datasets would certainly help in foster research on code review automation and would allow to obtain more reliable assessment of their performance. However, identifying these instances automatically is extremely challenging.

On the other side, a manual screening of hundreds of thousands of instances (the order of magnitude of these datasets) is an effort that cannot be embraced by a single research group. For these reasons, we believe that a community effort is needed in this direction. We are currently working on the creation of a platform to continuously mine code review data from GitHub, also featuring the possibility to manually check the mined instances and report the problematic ones with a simple click. The platform will be inspired by the recent DL4SE tool [dl4] we released in our research group. We hope that the availability of such a platform will help in creating a high-quality code review dataset which can be used by the research community.

7.1.2 Enriching the Contextual Information Provided to the Model

Another important limitation of our approaches, is the limited context that is provided as input to the model, based on which the prediction must be generated. As of now the view of the models is limited to a single method. This means that if a change affects several methods or even several files, the model is only aware of part of the change to review. Intuitively, more contextual information provided to the model could improve its performance, as also suggested by the recent work from the Microsoft group [LLG+22] which provides the model with diff hunks, possibly spanning across different methods. However, increasing the provided contextual information does also mean pushing up the input size with a consequent higher training cost and, likely, larger model required. Thus, we plan to investigate techniques for prioritizing and summarizing the contextual information surrounding a given code change, with the goal of providing the model with a compact yet expressive representation of the change to review.

7.1.3 Automated Solutions Targeting the Most Challenging Scenarios

In Chapter 6 we manually analyzed the correct and wrong predictions generated by state-of-the-art techniques, classifying them based on the type of code changes they implemented/required. This resulted in two taxonomies (Figures 6.1 and 6.2) pointing to specific code change types which code review automation techniques fail to automate.

It is possible that these types of changes are not well represented in the training dataset or that they are just more difficult to learn. In both cases, it may be worthwhile exploring the possibility to train specialized models focused on a specific type of change. We would like to investigate whether a family of specialized models used in combinations could provide a better support for code review automation as compared to the "single general model" approach currently adopted in the literature.

7.1.4 Project-specific and Developer-specific Recommendations

All the proposed models for code review automation are trained on data coming from thousands of online repositories, mixing together different coding styles and conventions. This could lead to a generic model which, however, has a limited usefulness in a real scenario, in which it provides some "generic" code recommendations without adapting to the specific

context in which it is used. Specializing the model to a single project or even to a single developer by adopting their coding style when revising code may improve the usefulness of the generated recommendations. For example, a code recommendation featuring an API already used in the past by the developer may be easier for them to understand as compared to one using an unknown API.

We plan to explore methodologies to embed such information (*e.g.,* the developer "knowledge" and coding style) in the model's input, studying its impact on the generated recommendations.

7.1.5 Identifying the Accuracy Breakeven Point for a Successful Technological Transfer

For the empirical evaluation of the code review automation techniques proposed in the literature, it is clear that none of them is currently ready for an adoption by developers (and, probably, also to run a large-scale evaluation with developers). However, it is unclear where the breakeven point is or, in other words, at which level of performance the the automation approach can be considered as a valid support for developers rather than a tool mostly bothering them with wrong recommendations.

While the answer to this question may appear simple to identify by, for example, running a survey with developers (*e.g.,* they could claim that above 50% of correct recommendations they would use the approach), our research in Chapter 6 showed that the accuracy alone only tells a small part of the story. Indeed, the techniques may be successful in automating/suggesting very simple code changes or very complex ones which, in turn, result in a substantially different amount of time saved to the developer.

Thus, the question is not only about which percentage of instances in the test set can be successfully automated, but also about which of those instances would mostly benefit developers. We plan to investigate this through user studies in which developers are supported (or not) by automated techniques having a different accuracy and are successful on different types of review tasks (*i.e.,* having a different complexity).

7.1.6 A Novel Way to Evaluate the Correctness of the Prediction

A major lesson learned while working on this research is that the quantitative metrics we use to assess the capabilities of the DL models usually underestimate the capabilities of the trained models. Especially when it comes to evaluate comments generated in natural language. Indeed, comparing the generated comment against the ground truth makes a prediction correct only if it uses exactly the same wording as the ground truth. However, as we discussed, it is possible to express the same concept using different wordings.

One very simple attempt to solve such a problem could be to use large language models such as ChatGPT to assess whether a comment automatically generated by the model is semantically equivalent to the one written by the human reviewer.

In general, we believe that better metrics are needed to quantitatively assess code review automation techniques.

7.1.7 Deepen the Evaluation of Large Language Models for Code Review Automation

In Section 6.2.3 we evaluated ChatGPT for the automation of the code review tasks targeted in this thesis. However, such a comparison was limited to a small sample of instances and, in a recent study we are conducting (not documented in this thesis), we found that open source developers are actually using ChatGPT for automating code review. This calls for a larger and deeper empirical evaluation aimed at better assessing large language models for these tasks, for example by experimenting with different prompts or by providing it with additional contextual information in a few-shot learning fashion.

7.2 Closing Words

In conclusion, with our research we have attempted the automation of code review tasks using DL, with the idea of imitating human developers. The goal of this line of research is not to replace human developers, but rather to support them and save some of the time they spend on code review. As we have discussed, there is still a lot of room for improvement which make code review automation a hot research topic nowadays.

Appendices

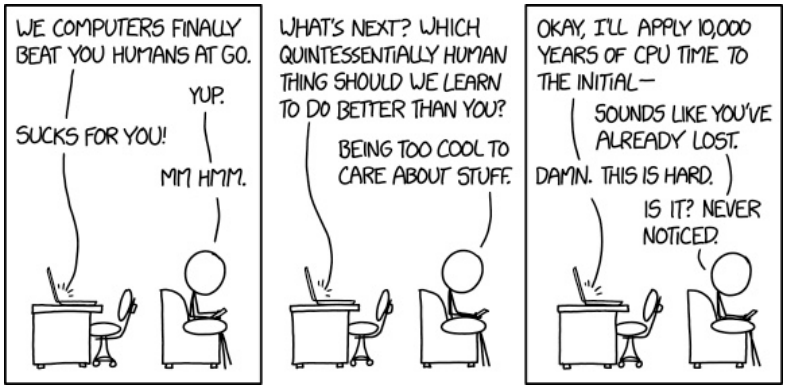
In this appendix we report other contributions in SE that do not fall within the specific topic of this thesis.

Appendix A presents the research done in the context of video game testing. We applied reinforcement learning to train an agent able to play and, at the same time, search for area of the game in which there is a drop in frame per seconds.

Appendix B describes an approach able to recommend how to replace custom implementations with open source APIs.

A

Using Reinforcement Learning for Load Testing of Video Games



Different from what happens for most types of software systems, testing video games has largely remained a manual activity performed by human testers. This is mostly due to the continuous and intelligent user interaction video games require. Recently, reinforcement learning (RL) has been exploited to partially automate functional testing. RL enables training *smart* agents that can even achieve super-human performance in playing games, thus being suitable to explore them looking for bugs. We investigate the possibility of using RL for load testing video games. Indeed, the goal of game testing is not only to identify functional bugs, but also to examine the game's performance, such as its ability to avoid lags and keep a minimum number of frames per second (FPS) when high-demanding 3D scenes are shown on screen. We define a methodology employing RL to train an agent able to play the game as a human while also trying to identify areas of the game resulting in a drop of FPS. We demonstrate the feasibility of our approach on three games. Two of them are used as proof-of-concept, by injecting artificial performance bugs. The third one is an open-source 3D game that we load test using the trained agent showing its potential to identify areas of the game resulting in lower FPS.

The content of this chapter has been presented in the following paper:

Using Reinforcement Learning for Load Testing of Video Games
Rosalia Tufano, Simone Scalabrino, Luca Pascarella, Emad Aghajani, Rocco Oliveto, Gabriele Bavota. In *Proceedings of the 44th International Conference on Software Engineering (ICSE 2022)*, pp. 2303–2314.

A.1 Introduction

The video game market is expected to exceed $200 billion in value in 2023 [mar]. In such a competitive market, releasing high-quality games and, consequently, ensuring a great user experience, is fundamental. However, the unique characteristics of video games (from hereon, games) make their quality assurance process extremely challenging. Indeed, besides inheriting the complexity of software systems, games development and maintenance require a diverse set of skills covered by many stakeholders, including graphic designers, story writers, developers, AI (Artificial Intelligence) experts, etc.

Also, games can hardly benefit from testing automation techniques [PPPB18], since even just exploring the total space available in a given game level requires an *intelligent* interaction with the game itself. For example, in a racing game, identifying a bug that manifests when the finish line is crossed requires a player able to successfully drive the car for the whole track (*i.e.,* requires the ability to drive the car). Thus, random exploration is not a viable option here.

Therefore, it comes without surprise that game testing is largely a manual process. Zheng *et al.* [ZXS+19] report that 30 human testers were employed for testing one of the games used in their study. Also, the challenges in testing games have been stressed by Lin *et al.* [LBH16], who showed that 80% of the 50 popular games they studied have been subject to urgent updates.

To support developers with game testing, researchers have proposed several techniques. These include approaches to test the stability of game servers [JLS+05, BJK06, CLS+10], model-based testing [IIKM15] using domain modeling for representing the game and UML state machines for behavioral modeling, as well as techniques specifically designed for testing board games [SNM09, dMSLTN17]. When looking at recent techniques aimed at proposing more general testing frameworks, those exploiting Reinforcement Learning (RL) are on the rise. This is due to the remarkable results achieved by RL-based techniques in playing games with super-human performance reported in the literature [BKM+19, AI19, HMVH+18, MKS+13, MKS+15, VEB+17].

RL is a machine learning technique aimed to train *smart* agents able to interact with a given environment (*e.g.,* a game) and to take decisions to achieve a goal (*e.g.,* win the game). RL is based on the simple idea of trial and error: The agent performs actions in the environment (of which it only has a partial representation) and receives a *reward* that allows it to assess its past actions/behavior with respect to the desired goal.

Recently, researchers started using RL not only to play games but also to test them and, in general, to improve their quality. The common idea behind these approaches is to reduce the human effort in playtesting using intelligent agents. RL-based agents have been used to help game designers in balancing crucial parameters of the game (*e.g.,* power-up item effects) [ZBB+19, PLV+20, ZFR14] and in testing the game difficulty [GEP+18, SNMB20]. Also, RL-based agents have been used to look for bugs in games [PSM17, BGTG20, ZXS+19, ABS21].

While agents are usually trained to play a game with the goal of winning, the aforementioned works trained the agent to not only advance in the game but also to explore it to search for bugs. For example, Ariyurek *et al.* [ABS21] combine RL and Monte Carlo Tree Search to find issues in the behavior of a game, given its design constraints and game scenario graph. The *ICARUS* framework [PSM17] is able to identify crashes and blockers bugs (*e.g.,* the game get stuck for a certain amount of time) while the agent is playing. Similarly, the approach by Zheng *et al.* [ZXS+19], also exploiting RL, can identify bugs spotted by the agent during training (*e.g.,* crashes). While these approaches pioneered the use of RL for game testing, they are mostly aimed at testing functional (*e.g.,* finding crashes) or design-related (*e.g.,* level design) aspects. However, these are not the only types of bug developers look for in playtesting. In a recent survey, Politowski *et al.* [PPG21] reported that for two out of the five games they considered (*i.e., League of Legends* by Riot and *Sea of Thieves* by Rare) developers partially automated game performance checks (*e.g.,* frame-rate). Similarly, Naughty Dog used specialized profiling tools[1] for finding which parts of a given scene caused a drop in the number of frames per second (FPS) in *The Last of Us*. Truelove *et al.* [TdAA21] report that game developers agree that *Implementation response* problems may severely impact the game experience.

Despite such a strong evidence about the importance of detecting performance issues in video games, to the best of our knowledge no previous work introduced automated approaches for load testing video games. We present *RELINE* (**R**einforcement l**E**arning for **L**oad test**IN**g gam**E**s), an approach exploiting RL to train agents able to play a given game while trying to load test it with the goal of minimizing its FPS. The agent is trained using a *reward* function enclosing two objectives: The first aims at teaching the agent how to advance in the game. The second rewards the agent when it manages to identify areas of the game exhibiting low FPS. The output of *RELINE* is a report showing areas in the game being negative outliers in terms of FPS, accompanied by videos of the gameplays exhibiting the issue. Such "reports" can simplify the identification and reproduction of performance issues, that are often reported in open-source 3D games (see *e.g.,* [dwac, 3dc, geo, dwab]) and that, in some cases, are challenging to reproduce (see *e.g.,* [dwaa, dwad]).

We experiment *RELINE* with three games. The first two are simple 2D games that we use as a proof-of-concept. In particular, we injected in the games artificial "performance bugs" [DPSSMB21] to check whether the agent is able to spot them. We show that the agent trained using *RELINE* can identify the injected bugs more often than (i) a random agent, and (ii) a RL-based agent only trained to play the game. Then, we use *RELINE* to load test an open-source 3D game [sup], showing its ability to identify areas of the game being negative outliers in terms of FPS.

[1] `https://youtu.be/yH5MgEbBOps?t=3494`

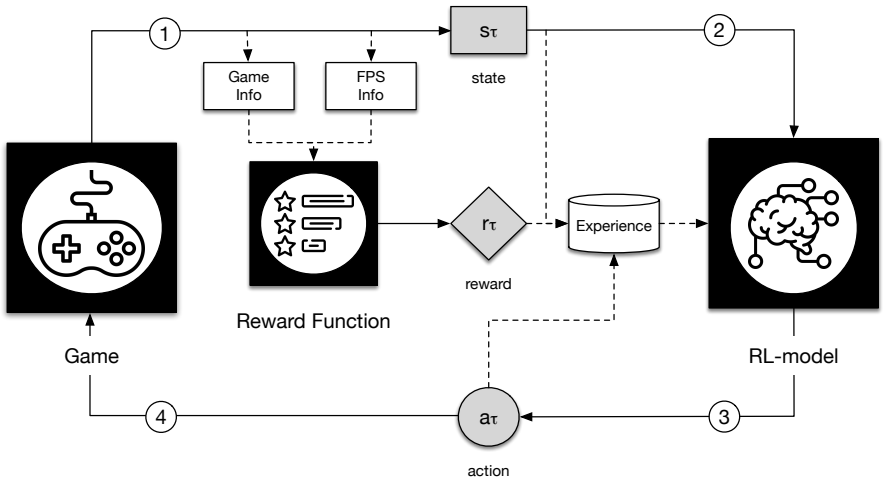


Figure A.1. *RELINE* overview

A.2 Approach

In this section we explain, from an abstract perspective, the idea behind *RELINE*. We describe in the study designs how we instantiated *RELINE* to the different games we experiment with (*e.g.,* details about the adopted RL models).

RELINE requires three main components: the *game* to load test, a *RL model*, representing the agent that must learn how to play the game while load testing it, and a *reward function*, used to reward the agent so that it can evaluate the worth of its actions for reaching the desired goal (*i.e.,* playing while load testing). The *RL model* is trained through the 4-step loop depicted in Fig. A.1 (see the circled numbers). The continuous lines represent steps performed at each iteration of the loop, while the dashed ones are only performed after a first iteration has been run (*i.e.,* after the agent performed at least one action in the game). When the first episode (*i.e.,* a run of the game) of the training starts (step 1), at each time step τ the game provides its state s_τ. Such a state can be, for example, a set of frames or a numerical vector representing what is happening in the game (*e.g.,* the agent's position). The *RL model* takes as input s_τ (step 2) and provides as output the action a_τ to perform in the game (step 3). When the agent has no experience in playing the game at the start of the training, the weights of the neural network in the RL model are randomly initialized, producing random actions. The action a_τ is executed in the game (step 4), which, in turn, generates the subsequent state $s_{\tau+1}$.

After the first iteration (*i.e.,* after having received at least one a_τ), the game also produces, at each iteration, the data needed to compute the reward function. In *RELINE* we collect (i) the information needed to assess how well the agent is playing the game (*e.g.,* time since the episode started and the episode score), and (ii) the FPS at time τ. It is required that the game developer instruments the game and provide APIs through which *RELINE* can acquire such pieces of information. We assume that this requires a minor effort.

The *reward function* aims at training an agent that is able to (i) play the game, thanks to the information indicating how well the agent is playing, and (ii) identify low-FPS areas, thanks to the information about the FPS. The output of the *reward function* is a number representing the reward obtained by the agent at time τ. In *RELINE*, the reward function for a given action is composed of two sub-functions: A *game reward function*, depending on how good the action is in the game (rg_τ), and a *performance reward function*, depending on how the action impacts the performance (rp_τ).

We combine such functions in $r_\tau = rg_\tau + rp_\tau$. The game reward function clearly depends on the game under test: A function designed for a racing game likely makes no sense for a role-playing game. In general, defining the reward function for learning to play should be performed by considering (i) what the goal of the game is (*e.g.,* drive on a track), and (ii) which information the game provides about the "successful behavior of the player" (*e.g.,* is there a score?). Even if less intuitive, the performance reward function is game-dependent as well: Assuming a tiny FPS drop (*e.g.,*-1%), the reward can be small for a role-playing game, in which it likely does not affect the whole experience, while it should be high for an action game, in which it could even cause the (unfair) player's defeat. Unlike the game reward function, we expect however minor changes to be required to adapt the performance reward function to a different video game (*i.e.,* tuning of the thresholds to use).

The state s_τ, the action a_τ, and the reward r_τ are then stored in an experience buffer. When enough experience has been accumulated, it is used to update the network weights. How experience is stored and used to update the network depends on the used RL model.

The episode ends when a final state is reached. Again, the definition of the final state depends on the game, and it could be based on a timeout (*e.g.,* each episode lasts at most 90 seconds) or on a specific condition that must be met (*e.g.,* the agent crosses the finish line). Once the episode ends, the game is reinitialized and the loop restarts. The training is performed for a number of episodes sufficient to observe a convergence in the total reward achieved by an agent during an episode (*e.g.,* if the trained agent obtains a reward of 100 for ten consecutive episodes the training is stopped).

A.3 Preliminary Study: Injecting Artificial Performance Issues

This preliminary study aims at assessing the ability of *RELINE* in identifying artificial "performance bugs" [DPSSMB21] we simulate in two 2D games. It is important to highlight that the *goal* of this study is only to demonstrate the applicability of *RELINE* for load testing games as a proof-of-concept. A case study on a 3D open-source game is presented in Section A.4.

A.3.1 Study Design

We select two 2D games, CartPole [Car] and MsPacman [Pac]. The former — Fig. A.2-(a) — is a dynamic system in which an unbalanced pole is attached to a moving cart, and the player must move the cart to balance the pole and keep it in a vertical position. The player loses if the pole is more than 12 degrees from vertical or the cart moves too far from the center.

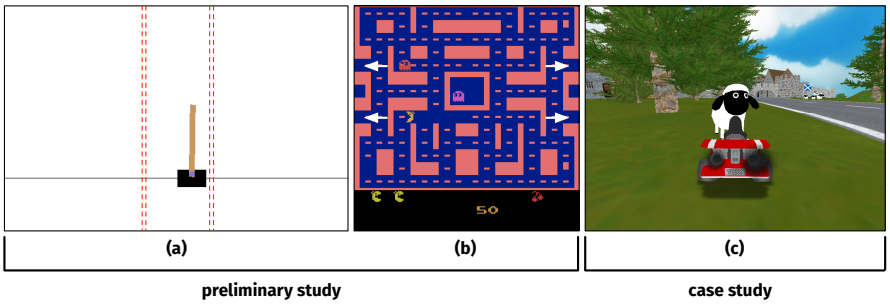


Figure A.2. Screenshots of games used in the preliminary study—Section A.3 (a) CartPole and (b) MsPacman, and in the case study—Section A.4 (c) SuperTuxKart.

The latter — Fig. A.2-(b) — is the classic Pac-Man game in which the goal is to eat all dots without touching the ghosts. Both games employ simple 2D graphics which bound the player's possible moves in only one (*e.g.,* left and right, for CartPole) or two (*e.g.,* left, right, up, and down, for MsPacman) dimensions. This is one of the reasons we selected these games for assessing whether a RL-based agent that learned how to play them can also be trained to look for artificial "performance bugs" we injected. Also, both games are integrated in the popular GYM Python toolkit [Gym] developed by OpenAI [BCP+16].

GYM can be used for developing and comparing RL-based agents in playing games. It acts as a middle layer between the *environment* (the game) and the *agent* (a virtual player). In particular, GYM collects and executes *actions* (*e.g.,* go left, go right) generated by the agent and returns to it the new state of the *environment* (*i.e.,* screenshots) with additional information such as the score in the episode. GYM comes with a set of integrated arcade games including the two we used in this preliminary study.

Bug Injection

We injected two artificial "performance bugs" in CartPole and four in MsPacman. The idea behind them is simple: When the agent visits specific areas for the first time during a game, the bugs reveal themselves (simulation of heavy resource loading). A natural way of achieving this goal would have been to introduce the bugs in the source code of the game and to implement the logic to spot FPS drops in the agent accordingly. This, however, would have slowed down the training of the agent. Therefore, we chose to use a more practically sound approach, inspired by the simulation of Heavy-Weight Operation (HWO) operator for performance mutation testing [DPSSMB21]: We directly assume that the agents observe the bugs when they visit the designated areas and act accordingly.

In CartPole, the agent can only move on the x axis (*i.e.,* left or right). When the game starts, the agent is in position $x = 0$ (*i.e.,* center of the axis) and it can change its position towards positive (by moving right) or negative (left) x values. The two bugs we injected manifest when $x \in [-0.50, -0.45]$ and $x \in [0.45, 0.50]$ — dashed lines in Fig. A.2-(a). We use intervals rather than specific values (*e.g.,*-0.45) because the position of the agent is a float: if it moves to position -0.450001, we want to reward it during the training for having

found the injected bug. Concerning MsPacman, we assume that a performance bug manifests when the agent enters the four *gates* indicated by the white arrows in Fig. A.2-(b).

As detailed in Section A.3.1, we assess the extent to which *RELINE* is able to identify the bugs we injected while playing the games. To have a baseline, we compare its results with those of a RL-based agent only trained to play each of the two games (from hereon, *rl-baseline*), and with a *random agent*. Since *RELINE* will be trained with the goal of identifying the bugs (details follow), we expect it to adapt its behavior to not only successfully play the game, but to also exercise more often the "buggy" areas of the games.

Learning to Play: RL Models and Game Reward Functions

We trained the *rl-baseline* agent (*i.e.,* the one only trained to learn how to play) for CartPole using the *cross-entropy method* [RK04] as RL model. We choose this method because, despite its simplicity, it has been shown to be effective in applications of RL to small environments such as CartPole [Lap18].

The core of the cross-entropy method is a feedforward neural network (FNN) that takes as input the state of the game and provides as output the action to perform. The state of the game for CartPole is a vector of dimension 4 containing information about the x coordinate of the pole's center of mass, the pole's speed, its angle with respect to the platform, and its angular speed. There are two possible actions: go right, go left. Once initialized with random weights, the agent (*i.e.,* the FNN) starts playing while retaining the experience acquired in each episode: The experience is represented by the state, the action, and the reward obtained during each step of the episode.

The goal is to keep the pole in balance as long as possible or until the maximum length of an episode (that we set to 1,000 steps) is reached. The *game reward function* is defined so that the agent receives a +1 reward for each step it manages to keep the pole balanced. The total score achieved is also saved at the end of each episode. After $n = 16$ consecutive episodes the agent stops playing, selects the $m = 11$ (70%) episodes having the highest score, and uses the experience in those episodes to update the weights of the FNN (n and m have been set according to [Lap18]).

Instead, we trained the *rl-baseline* agent for MsPacman using a Deep Q Network (DQN) [MKS+13]. In our context, a DQN is a Convolutional Neural Network (CNN) that takes as input a set of contiguous screenshots of the game (in our case 4, as done in previous works [MKS+13, MKS+15]) representing the state of the game and returns, for each possible action defined in the game (five in this case: go up, go right, go down, go left, do nothing), a value indicating the expected reward for the action given the current state (Q value). The multiple screenshots are needed to provide more information to the model about what is happening in the game (*e.g.,* in which direction the agent is moving). The goal of the DQN is the same as the FNN: selecting the best action to perform to maximize the reward given the current state. Differently from the previous model, the DQN is updated not on entire episodes but by randomly batching "experience instances" among 10k steps saved during the most recent episodes. An "experience instance" is saved after each step τ, and is represented by the quadruple $(s_{\tau-1}, a_\tau, s_\tau, r_\tau)$, where $s_{\tau-1}$ is the input state, a_τ is the action selected by the

agent, s_τ is the resulting state obtained by running a_τ in $s_{\tau-1}$ and r_τ is the received reward.

The CNN is initialized with random weights, and the agent starts playing while retaining the experience of each step. When enough experience instances have been collected (10k in our implementation [Lap18]), the CNN starts updating at each step selecting a random batch of experience instances. The reward function for MsPacman provides a +1 reward every time the agent eats one of the dots and a 0 reward otherwise.

Instantiating *RELINE*: Performance Reward Functions

To train *RELINE* to play while looking for the injected bugs, we use a simple *performance reward function*: In both the games, we give a reward of +50 every time the agent, during an episode, spots one of the injected artificial bugs. As previously mentioned, the bugs reveal themselves only the first time the agent visits each buggy position; this means that the performance-based reward is given at most twice for CartPole and four times for MsPacman.

Data Collection and Analysis

We compare *RELINE* against the two previously mentioned baselines: *rl-baseline* and the *random agent*. Both *RELINE* and *rl-baseline* have been trained for 3,200 episodes on CartPole and 1,000 on MsPacman. The different numbers are due to differences in the games and in the RL model we exploited. In both cases, we used a number of episodes sufficient for *rl-baseline* to learn how to play (*i.e.,* we observed a convergence in the score achieved by the agent in the episodes).

Once trained, the agents have been run on both games for additional 1,000 episodes, storing the performance bugs they managed to identify in each episode. Since different trainings could result in models playing the game following different strategies, we repeated this process ten times. This means that we trained 10 different models for both *RELINE* and *rl-baseline* and, then, we used each of the 10 models to play additional 1,000 episodes collecting the spotted performance bugs. Similarly, we executed *random agent* 10 times for 1,000 episodes each. In this case, no training was needed.

We report descriptive statistics (mean, median, and standard deviation) of the number of performance bugs identified in the 1,000 played episodes by the three approaches. A high number of episodes in which an approach can spot the injected bugs indicate its ability to look for performance bugs while playing the game.

A.3.2 Preliminary Study Results

Table A.1 shows for each of the two games (CartPole and MsPacman) the number k of artificial bugs we injected and, for each of the three techniques (*i.e., RELINE*, *rl-baseline*, and the *random agent*), descriptive statistics of the number of episodes (out of 1,000) they managed to identify at least n of the injected bugs, with n going from 1 to k at steps of 1.

For both games, it is easy to see that the *random agent* is rarely able to identify the bugs. Indeed, this agent plays without any strategy as it is able to identify bugs only by chance in a few episodes out of the 1,000 it plays. This is also due to the fact that the *random agent*

Table A.1. Number of episodes (out of 1,000) in which *RELINE*, *rl-baseline*, and the *random agent* identify the injected bugs.

Game	#Bugs found	*RELINE*			*rl-baseline*			*random agent*		
		mean	median	stdev	mean	median	stdev	mean	median	stdev
CartPole	1/2	965	984	47	715	706	107	12	11	4
	2/2	102	47	177	5	1	7	0	0	0
MsPacman	1/4	971	989	59	700	680	228	24	23	5
	2/4	966	985	63	356	343	169	17	16	3
	3/4	914	941	87	114	80	90	1	1	1
	4/4	879	907	106	25	23	17	1	1	1

quickly looses the played episodes due to its inability to play the game. This confirms that these approaches are not suitable for testing video games.

Concerning CartPole, both *RELINE* and *rl-baseline* are able to spot at least one of the two bugs in several of the 1,000 episodes. The median is 984 for *RELINE* and 706 for *rl-baseline*. The success of *rl-baseline* is soon explained by the characteristics of CartPole: Considering where we injected the bugs — see Fig. A.2-(a) — by playing the game it is likely to discover at least one bug (*e.g.,* if the player tends to move towards left, the bug on the left will be found). What it is instead unlikely to happen by chance is to find both bugs within the same episode. We found that it is quite challenging, even for a human player, to move the cart first towards one side (*e.g.,* left) and, then, towards the other side (right) without losing due to the pole moving more than 12 degrees from vertical. As it can be seen in Table A.1, *RELINE* succeeds in this, on average, for 102 episodes out of 1,000 (median 47), as compared to the 5 (median 1) of *rl-baseline*. This indicates that *RELINE* is pushed by the reward function to explore the game looking for the injected bugs, even if this makes playing the game more challenging. Similar results have been achieved on MsPacman.

In this case, the DQN is effective in allowing *RELINE* to play while exercising the points in the game in which we injected the bugs. Indeed, on average, *RELINE* was able to spot all four injected bugs in 879 out of the 1,000 played episodes (median=907), while *rl-baseline* could achieve such a result only in 25 episodes.

Summary of the Prelimiary Study. *RELINE* allows obtain agents able not only to effectively play a game but also to spot performance issues. Compared to *rl-baseline*, the main advantage of *RELINE* is that it identifies bugs more frequently while playing.

A.4 Case Study: Load Testing an Open Source Game

We run a case study to experiment the capability of *RELINE* in load testing an open-source 3D game. Differently from our preliminary study (Section A.3), we do not inject artificial bugs. Instead, we aim at finding parts of the game resulting in FPS drops.

A.4.1 Study Design

For this study, we use a 3D kart racing game named *SuperTuxKart* [sup] — see Fig. A.2-(c). This game has been selected due to the following reasons. First, we wanted a 3D game in which, as compared to a 2D game, FPS drops are more likely because of the more complex rendering procedures. Second, SuperTuxKart is popular open-source project that counts, at the time of writing, over 3k stars on GitHub. Third, it is available an open-source wrapper that simplifies the implementation of agents for SuperTuxKart [PyS].

The existence of a wrapper like the one we used is crucial since it allows, for example, to advance in the game frame by frame (thus simplifying the generation of the inputs to the RL model), to execute actions (*e.g.,* throttle or brake), and to acquire game internals (*e.g.,* kart centering, distance to the finish line). Also, using this wrapper, it is possible to compute the time needed by the game to render each frame and, consequently, calculate the FPS. Finally, the wrapper allows to have simplified graphics (*e.g.,* removing particle effects, like rain, that could make the training more challenging).

Learning to Play: RL Models and Game Reward Functions

The training of the *rl-baseline* agent has been performed using the DQN model previously applied in MsPacman.

We use the previously mentioned *PySuperTuxKart* [PyS] to make the agent interact with the game. For the sake of speeding up the training, the screenshots extracted from the game have been resized to 200x150 pixels and converted in grayscale before they are provided as input to the model. Moreover, as previously done for MsPacman, multiple (four) screenshots are fed to the model at each step. Thus, the representation of the state of the game provided to the model is a 4×200×150 tensor. The details of the model and its implementation are available in our replication package [repg].

A critical part of the learning process is the definition of the *game reward function*. Being SuperTuxKart a racing game, an option could have been to penalize the agent for each additional step required to finish the game. Consequently, to maximize the final score, the agent would have been pushed to reduce the number of steps and, therefore, to drive as fast as possible towards the finish line. However, considering the non-trivial size of the game space, such a reward function would have required a long training time. Thus, we took advantage of the information that can be extracted from the game to help the agent in the learning process.

SuperTuxKart provides two coordinates indicating where the agent is in the game: `path_done` and `centering`.

The former indicates the path traversed by the agent from the starting line of the track, while the latter represents the distance of the agent from the center of the track. In particular, `centering` equals 0 if the agent is at the center of the track, and it moves away from zero as the agent moves to either side: going towards right results in positive values of the `centering` value, going left in negative values. We indicate these coordinates with x (`centering`) and y (`path_done`), and we define δ_y as the path traversed by the agent in a specific step: Given y_i the value for `path_done` at step i, we compute δ_y as $y_i - y_{i-1}$.

Basically, δ_y measures how fast the agent is advancing towards the finish line.

Given x and δ_y for a given step i, we compute the reward function as follows:

$$rg_i = \begin{cases} -1 & \text{if } |x| > \theta \\ \max(\min(\delta_y, M), 0) & \text{otherwise} \end{cases}$$

First, if the agent goes too far from the center of the track ($|x| > \theta$), we penalize it with a negative reward. Otherwise, if the agent is close to the center ($|x| \leq \theta$), we can have two scenarios: (i) if it is not moving towards the finish line ($\delta_y \leq 0$), we do not give any reward (the minimum reward is 0); (ii) if it is moving in the right direction ($\delta_y > 0$), we give a reward proportional to the speed at which it is advancing (δ_y), up to a maximum of M.

In our experimental setup, we set $\theta = 20$ because it roughly represents the double of $|x|$ when the agent approaches the sides of the road in the level, and $M = 10$ as it is the same maximum reward also given by the *performance reward function*, as we explain below. Finally, we reward the agent when it crosses the finish line with an additional $+1,000$ bonus.

Instantiating *RELINE*: Performance Reward Function

To define the *performance reward function* of *RELINE* for SuperTuxKart, the first step to perform is to define a way to reliably capture the FPS of the game during the training. In this way, we can reward the agent when it manages to identify low-FPS points. As previously said, we use PySuperTuxKart to interact with the game and such a framework keeps the game frozen while the other instructions of *RELINE* (*e.g.,* the identification of the action to execute) are run. Since the framework runs the game in the same process in which we run *RELINE* and since we do not use threads, we can safely use a simple method for computing the time needed to render the four frames: We get the system time before (T_{before}) and after (T_{after}) we trigger the rendering of the frames and we compute the time needed at step i as $rT_i = T_{after} - T_{before}$. Such a value is negatively correlated with the FPS (higher rendering time means lower FPS).

The *performance reward function* we use is the following:

$$rp_i = \begin{cases} 10 & \text{if } |x| \leq \theta \wedge rT_i > t \\ 0 & \text{otherwise} \end{cases}$$

We give a performance-based reward of 10 when the agent takes more than t milliseconds to render the frames at a given point (causing an FPS drop). We explain the tuning of t later. We do not give such a reward when $|x| > \theta$ (the kart is far from the center) since we want the agent to spot issues in positions that are likely to be explored by real players (*i.e.,* reasonably close to the track).

Finally, in *RELINE* we do not give a fixed $+1,000$ bonus reward when the agent crosses the finish line but we assign a bonus computed as $10 \times \sum_{i=1}^{steps} rp_i$, *i.e.,* proportional to the total performance-based reward accumulated by the agent in the episode. This is done to push the agent to visit more low-FPS points during an episode.

Data Collection and Analysis

As done in our preliminary study, we compare *RELINE* with *rl-baseline* (*i.e.,* the agent only trained to play the game) and with a *random agent*.

Training *rl-baseline* and *RELINE*. While we used different reward functions for the two RL agents, we applied the same training process for both of them. We trained each model for 2,300 episodes, with one episode having a maximum duration of 90 seconds or ending when the agent crosses the finish line of the racing track (the agent is required to perform a single lap). We set the 90 seconds limit since we observed that, by manually playing the game, ∼70 seconds are sufficient to complete a lap. The 2,300 episodes threshold has been defined by computing the average reward obtained by the two agents every 100 episodes and by observing when a plateau was reached by both agents. We found 2,300 episodes to be a good compromise for both agents (graphs plotting the reward function are available in the replication package [repg]).

The trained *rl-baseline* agent has been used to define the threshold t needed for the *RELINE*'s reward function (*i.e.,* for identifying when the agent found a low-FPS point and should be rewarded).

In particular, once trained, we run *rl-baseline* for 300 episodes, storing the time needed by the game to render the subsequent four frames after every action recommended by the model.[2] This resulted in a total of 48,825 data points s_{FPS}, representing the standard FPS of the game in a scenario in which the player is only focused on completing the race as fast as possible.

Starting from the 48,825 s_{FPS} data points collected in the 300 episodes played by the trained *rl-baseline* agent, we apply the five-σ rule [Gra06] to compute a threshold able to identify outliers. The five-σ rule states that in a normal distribution (such as s_{FPS}) 99.99% of observed data points lie within five standard deviations from the mean. Thus, anything above this value can be considered as an outlier in terms of milliseconds needed to render the frames. For this reason, we compute $t_b = mean(s_{FPS}) + 5 \times sd(s_{FPS})$ as a candidate base threshold to identify low-FPS points. However, t_b cannot be directly used as the t value of our reward function. Indeed, we observed that the time needed for rendering frames during the *RELINE*'s training is slightly higher as compared to the time needed when the trained *rl-baseline* agent is used to play the game. This is due to the fact that the load on the server (and in particular on the GPU) is higher during training. To overcome this issue, we perform the following steps.

At the beginning of the training, we run 100 *warmup episodes* in which we collect the time needed to render the four frames after each action performed by the agent. Then, we compute the first (Q_1^{tr}) and the third (Q_3^{tr}) quartile of the obtained distribution and compare them to the Q_1 and Q_3 of the distribution obtained in the 300 episodes used to define t_b (*i.e.,* those played by the trained *rl-baseline* agent). During the *warmup episodes*, the agent selects the action to perform almost randomly (it still has to learn): Therefore, it would not be able to explore a substantial area of the game (*i.e.,* of the racing track), thus not

[2] Since we wanted to measure the frames rendering time in a standard scenario in which the agent was driving the kart, we stopped an episode if the agent got stuck against some obstacle.

providing a distribution of timings comparable with the ones obtained when the trained *rl-baseline* agent that played the 300 episodes. For this reason, during the 100 *warmup episodes* of the training, the action to perform is not chosen by the agent currently under training, but by the trained *rl-baseline* agent (*i.e.,* the same used in the 300 episodes). This does not impact in any way the load on the server that remains the one we have during the training of *RELINE* since the only change we have is to ask for the action to perform to the *rl-baseline* agent rather than to the one under training. However, the whole training procedure (*e.g.,* capturing the frames and updating the network) stays the same.

We compute the additional "cost" brought by the training in rendering the frames during the game using the formula $\delta = max(Q_1^{tr} - Q_1, Q_3^{tr} - Q_3)$. We use the first and third quartiles since they represent the boundaries of the central part of the distribution, *i.e.,* they should be quite representative of the values in it. We took as δ the maximum of the two differences to be more conservative in assigning rewards when the agent identifies low-FPS points. The final value t we use in our reward function when training *RELINE* to load test SuperTuxKart is defined as: $t = t_b + \delta = 18.36$.[3]

Thus, if *RELINE* is able, during the training, to identify a point in the game requiring more than t milliseconds to render four frames, then it receives a reward as explained in Section A.4.1.

The training of *rl-baseline* took ∼3 hours, while *RELINE* requires substantially more time due to the fact that, after each step performed by the agent, we collect and store information about the time needed to render the frames (this is done million of times). This pushed the training for *RELINE* up to ∼30 hours.

Reliability of Time Measurements. It is important to clarify that the FPS of the game can be impacted by the hardware specifications and the current load of the machine running it. In other words, running the same game on two different machines or on the same machine in two different moments can result in variations of the FPS. For this reason, all the experiments have been performed on the same server, equipped with 2 x 64 Core AMD 2.25GHz CPUs, 512GB DDR4 3200MHz RAM, and an nVidia Tesla V100S 32GB GPU. Also, the process running the training of the agents or the collection of the 48,825 s_{FPS} with the trained *rl-baseline* agent was the only process running on the machine besides those handled by the operating system (Ubuntu 20.04). On top of that, the process was always run using the `chrt -rr 1` option, that in Linux maximizes the priority of the process, reducing the likelihood of interruptions.

Despite these precautions, it is still possible that variations are observed in the FPS not due to issues in the game, but to external factors (*e.g.,* changes in the load of the machine). To verify the reliability of the collected FPS data, we run a constant agent performing always the same actions in the game for 300 episodes. The set of actions has been extracted from one of the episodes played by the *rl-baseline* agent, that was able to successfully conclude the race. Then, we plotted the time needed by the game to render the four frames following

[3]We identify as low-FPS points the ones in which the FPS is lower than 218. Such a number is still very high, more than enough for any human player, in practice. Note that we run the game using high-performance hardware and, most importantly, with the lowest graphic settings. The equivalent in normal conditions would be much lower.

each action made by the agent. Since we are playing 300 times exactly the same episode, we expect to observe the same trend in terms of FPS for each game. If this is the case, it means that the way we are measuring the FPS is reliable enough to reward the agent when low-FPS points are identified.

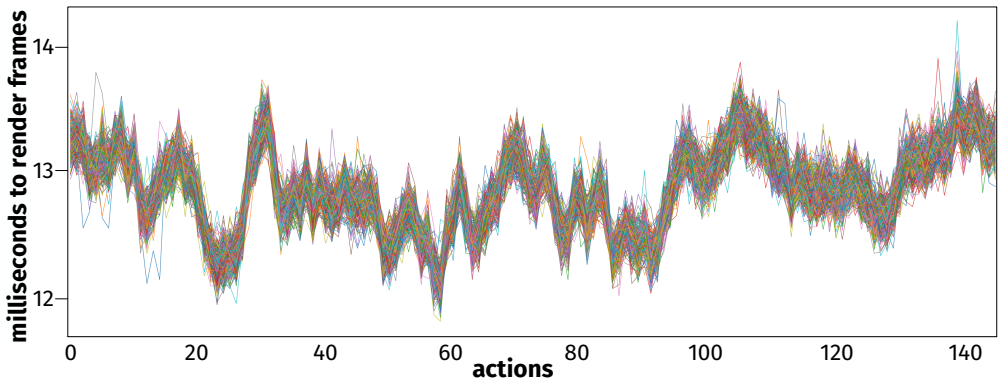


Figure A.3. Rendering times for 300 episodes (same actions).

Fig. A.3 shows the achieved results: The y-axis represents the milliseconds needed to render four frames in response to an agent's action (x-axis) performed in a specific part of the game. While, as expected, small variations are possible, the overall trend is quite stable: Points of the game requiring longer time to render frames are consistently showing across the 300 episodes, resulting in a clear trend. We also computed the Spearman's correlation [Spe04] pairwise across the 300 distributions, adjusting the obtained p-values using the Holm's correction [Hol79b].

We found all correlations to be statistically significant (adjusted p-values < 0.05) with a minimum ρ=0.77 (strong correlation) and a median ρ=0.91 (very strong correlation). This confirms the common FPS trends across the 300 episodes.

Running the Three Techniques to Spot Low-FPS Areas. After the 2,300 training episodes, we assume that both the RL-based agents learned how to play the game, and that *RELINE* also learned how to spot low-FPS points. Then, as also done in our preliminary study, we train both agents for additional 1,000 episodes, storing the time needed to render the frames in every single point they explored during each episode (where a point is represented by its coordinates, *i.e.,* `centering`=x and `path_done`=y). We do the same also with the *random agent*.

Data Analysis. The output of each of the three agents is a list of points with the milliseconds each of them required to render the subsequent frames. Since each agent played 1,000 episodes, it is possible that the same point is covered several times by an agent, with slightly different FPS observed (as previously explained, small variations in FPS are possible and expected across different episodes). We classify as low-FPS points those that required more than t milliseconds to render the four subsequent frames more than 50% of times they have been covered by an agent. This means that, if across the 1,000 episodes a point p is exercised 100 times by an agent, at least 51 times the threshold t must be exceeded to consider p as a low-FPS point.

In practice, a developer using *RELINE* for identifying low-FPS points could use a higher threshold to increase the reliability of the findings. However, for the sake of this empirical study, we decided to be conservative.

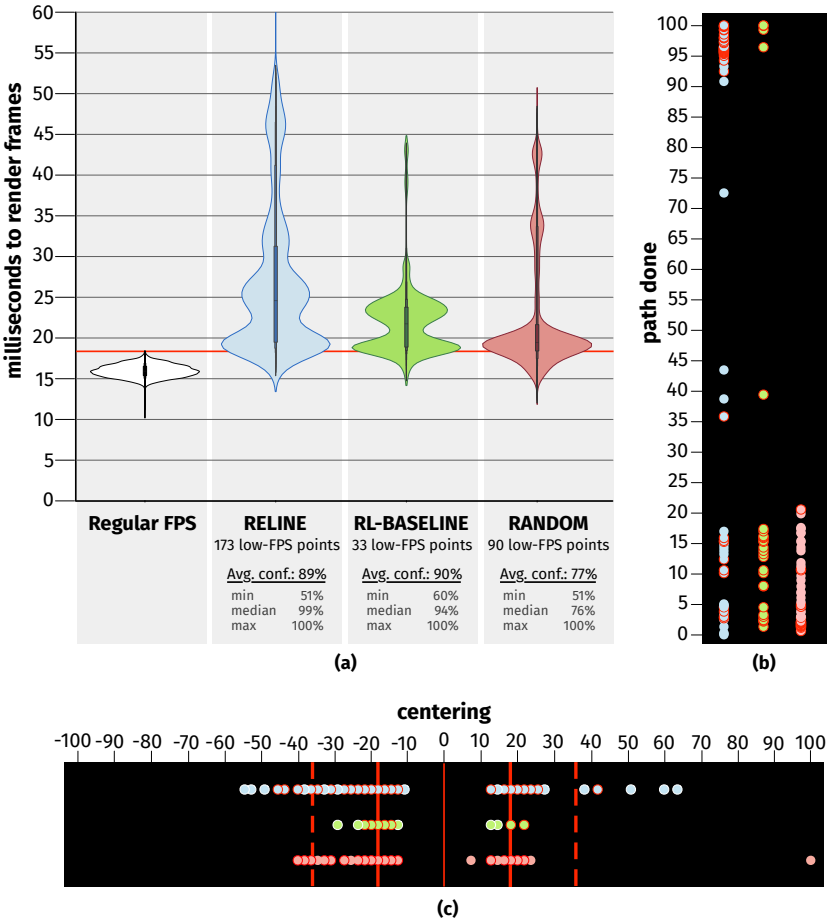


Figure A.4. Results of the study: (a) reports the distributions of timings for the low-FPS points with summary statistics, while (b) and (c) depict the *path done* and *centering* coordinates at which the such points were observed, respectively.

Then, we compare the characteristics of the low-FPS points identified by the three approaches. Specifically, we analyze: (i) how many different low-FPS points each approach identified; (ii) the number of times each low-FPS point has been exercised by each agent in the 1,000 episodes; (iii) the *confidence* of the identified points (*i.e.,* the percentage of times an exercised point resulted in low FPS). Given the low-FPS points identified by each agent, we draw violin plots showing the distribution of timings needed to render the frames when the agent exercised them (the higher the timings, the lower the FPS). We compare these distributions using Mann-Whitney test [Con98] with p-values adjustment using the Holm's correction [Hol79b].

We also estimate the magnitude of the differences by using the Cliff's Delta (d), a non-parametric effect size measure [GK05b] for ordinal data. 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 $|d| \geq 0.474$ [GK05b].

A.4.2 Study Results

Fig. A.4 summarizes the main findings of our case study. Fig. A.4-(a) shows the distribution of time needed to render the game frames (*i.e.,* our proxy for FPS) for four groups of points. The first violin plot on the left (*i.e.,* Regular FPS) shows the timing for points that have never resulted in a drop of FPS in any of the 3,000 episodes played by the three agents (1,000 each). These serve as baseline to better interpret the low-FPS points exercised by the agents. The other three violin plots show the distributions of timing for the low-FPS points identified by *RELINE* (blue), *rl-baseline* (green), and the *random agent* (red).

Below each violin plot we report the number of low-FPS points identified by each agent and descriptive statistics (average, median, min, max) of the confidence for the low-FPS points. A 100% confidence means that all times that a low-FPS point has been exercised in the 1,000 episodes played by the agent it required more than $t = 18.36$ milliseconds to render the subsequent frames. The t threshold is represented by the red horizontal line. On average, *RELINE* exercised each low-FPS point 89 times in the 1,000 episodes, against the 210 of *rl-baseline* and the 829 of the *random agent* (the same point can be exercised multiple times in an episode).

RELINE identified 173 low-FPS points, as compared to the 33 of *rl-baseline* and the 90 of the *random agent*. The confidence is similar for *RELINE* (median=99%) and *rl-baseline* (median=94%), while it is lower for the *random agent* (median=76%). Thus, the low-FPS points identified by the two RL-based agents are, overall, quite reliable. Concerning the number of low-FPS points identified, *RELINE* identifies more points as compared to *rl-baseline* (173 *vs* 33). This is expected since it has the explicit goal of load testing the game, However, what could be surprising at first sight is the high number of low-FPS points identified by the *random agent* (90). Fig. A.4-(b) and Fig. A.4-(c) help in interpreting this finding.

Fig. A.4-(b) plots the `path_done` (y coordinate) for each low-FPS point identified by each agent, using the same color schema of the violin plots (*e.g.,* blue corresponds to *RELINE*). If multiple points fall in the same coordinate (*i.e.,* same `path_done` but different `centering`), they are shown with a red border. The scale of the `path_done` has been normalized between 0 and 100, where 0 corresponds to the starting line of the track and 100 to its finish line. Similarly, Fig. A.4-(c) plots the `centering` (x coordinate) for the low-FPS points. The line at 0 represents the center of the track, while the continuous lines in position ∼-18 and ∼18 depict the limits of the track. Finally, the dashed lines represent the area of the game we asked *RELINE* to explore: based on our reward function, we penalize the agent for going outside the [-20, +20] range that, normalized, corresponds to ∼[-36, +36]. Also *rl-baseline* is penalized outside of this area.

As expected, the *random agent* is not able to advance in the game: The low-FPS points it identifies are all placed near the starting line — red dots in Fig. A.4-(b). This indicates that a random agent can be used to exercise a specific part of a game, but it is not able to explore the game as a player would do. This is also confirmed by the red dots in Fig. A.4-(c), with the *random agent* exploring areas of the game far from the track and that a human player is unlikely to explore. Also, it is worth noting that in SuperTuxKart each episode lasts (based on our setting) 90 seconds if the agent does not cross the finish line. However, as shown in our preliminary study, in other games such as MsPacman a *random agent* could quickly lose an episode without having the chance to explore the game at all.

The low-FPS points identified by *RELINE* (blue dots) and by *rl-baseline* (green) are instead closer to the track and, for what concerns *RELINE*, they are within or very close the area of the game we ask it to explore — see dashed lines in Fig. A.4-(c). Thus, by customizing the reward function, it is possible to define the area of the game relevant for the load testing.

Looking at Fig. A.4-(b), we can see that *RELINE* is also able to identify low-FPS in different areas of the game with, however, a concentration close to the beginning and the end of the game. It is difficult to explain the reason for such a result, but we hypothesize two possible explanations.

First, it is possible that the "central" part of the game simply features less low-FPS areas. This would also be confirmed by the fact that *rl-baseline* only found one low-FPS point in that part of the game. Also, the training and the reward function could have driven *RELINE* to explore more the starting and the ending of the game. The starting part is certainly the most explored since, at the beginning of the training, the agent is basically a random agent. Thus, it mostly collects experience about low-FPS points found in the beginning of the game since, similarly to the *random agent*, it is not able to advance in the game. It is important to remember that the data in Fig. A.4 only refers to the 1,000 games played by *RELINE* after the 2,300 training games, so we are not including the random exploration done at the beginning of the training in Fig. A.4. However, once the agent learns several low-FPS points in the starting of the game, it can exercise them again and again to get a higher reward.

Concerning the end of the game, we set a maximum duration of 90 seconds for each game, but we know that a well-trained agent can complete the lap in ∼70 seconds. It is possible that the agent used the remaining time to better explore the last part of the game before crossing the finish line, thus finding a higher number of low-FPS points in that area. Additional trainings, possibly with a different *reward function*, are needed to better explain our finding.

Concerning the violin plots in Fig. A.4-(a), we can see that *RELINE* and *rl-baseline* exhibit a similar distribution, with *RELINE* being able to identify some stronger low-FPS points (*i.e.,* longer time to render frames). All distributions have, as expected, the median above the t threshold, with *RELINE*'s one being higher (24.54 *vs* 21.69 for *rl-baseline* and 19.39 for *random agent*). The highest value of the distributions is 65.92 (60.7 FPS) for *RELINE*, against 44.81 (89.3 FPS) for *rl-baseline* and 50.73 (78.8 FPS) for *random agent*. Remember that all these values represent milliseconds to load frames after an action performed by the agents.

Table A.2. Results of Mann-Whitney test (adjusted p-value) and Cliff's Delta (d) when comparing the distributions of rendering times — boldface indicates higher times.

Test	p-value	OR
RELINE *vs rl-baseline*	<0.001	0.34 (Medium)
RELINE *vs random agent*	<0.001	0.36 (Medium)
rl-baseline *vs random agent*	<0.001	0.16 (Small)

Table A.2 shows the results of the statistical comparisons among the three distributions. In each test, the approach reported in boldface is the one identifying stronger low-FPS points (*i.e.,* more extreme points requiring longer rendering time for their frames). The adjusted p-values report a significant difference (p-value < 0.001) in favor of *RELINE* against both *rl-baseline* and the *random agent* (in both cases, with a medium effect size). Thus, the low-FPS points identified by *RELINE* tend to require longer times to render frames. Fig. A.2-(c) shows an example of low-FPS point identified by *RELINE*: Crashing against the sheep results in a drop of FPS.

Finally, it is worth commenting about the overlap of low-FPS points identified by the three agents. Indeed, *RELINE* and *rl-baseline* found 14 low-FPS points in common (*i.e.,* same x and y coordinates), while the overlap is of 11 points for *RELINE* and *random agent*, and 10 for *rl-baseline* and *random agent*. The most interesting finding of this analysis is that *rl-baseline* was able to identify only 19 points missed by *RELINE*, while the latter found 159 points missed by *rl-baseline*. This supports the role played by the *reward function* in pushing *RELINE* to look for low-FPS points.

Summary of the Case Study. *RELINE* is the best approach for finding low-FPS points in SuperTuxKart. A *random agent* is not able to spot issues that require playing skills, and *rl-baseline* only finds a small portion of the low-FPS points.

A.5 Threats to Validity

Threats to Construct Validity. The main threats to the construct validity of our study are related to the process we adopted in our case study (Section A.4) to identify low-FPS points. Based on our experiments, and in particular on the findings reported in Fig. A.3, our methodology should be reliable enough to identify variations in FPS. Still, some level of noise can be expected, and for this reason all our analyses have been run at least 300 times, while 1,000 episodes were played by each of the experimented approaches.

Concerning our preliminary study (Section A.3), it is clear that the bugs we injected are not representative of real performance bugs in the subject games. However, they are inspired from a performance mutation operator defined in the literature [DPSSMB21]. Our preliminary study only serves as a proof-of-concept to verify whether, by modifying the reward function, a RL-based agent would adapt its behavior to look for bugs while playing the game.

Threats to Internal Validity. In our case study, to ease the training we did not use the "real" game, but its wrapped version, *i.e.,* PySuperTuxKart [PyS]. While the core game is the same, the version we adopted does not contain the latest updates and it includes additional Python code that may affect the rendering time. We assume that such a time is constant for all the frames since it simply triggers the frame rendering operation in the core game. Besides, we forced the game to run with lowest graphics settings to speed up rendering: For example, we excluded dynamic lighting, anti-aliasing, and shadows. Therefore, the low-FPS points found in PySuperTuxKart may be irrelevant in the original game or with other graphic settings. Also, we applied the five-σ rule to define a threshold for defining what a low-FPS point is. The threshold we set might be not indicative of relevant performance issues.

Still, the goal of our study was to show that once set specific requirements (*e.g.,* the threshold t, the area to explore, etc.), the agent is able to adapt trying to maximize its reward. Thus, we do not expect changes in the threshold to invalidate our findings.

Threats to conclusion validity. In our data analysis we used appropriate statistical procedures, also adopting p-value adjustment when multiple tests were used within the same analysis.

Threats to External Validity Besides the proof-of-concept study we presented in Section A.3, our empirical evaluation of *RELINE* includes a single game. This does not allow us to generalize our findings. The reasons for such a choice lie in the high effort we experienced as researchers in (i) building the pipeline to interact with the game, (ii) finding and experimenting with a reliable way to capture the FPS, (iii) defining a meaningful reward function that allowed the agent to successfully play the game in the first place and, then, to also spot low-FPS points. These steps were a long trial-and-error process with the most time consuming part being the trainings needed to test the different reward functions we experimented before converging towards the ones presented in this paper. Indeed, testing a new version of a reward function required at least one week of work with the hardware at our disposal (including implementation, training, and data analysis).

This was also due to the impossibility of using multiple machines or to run multiple processes in parallel on the same server. Indeed, as explained, using the exact same environment to run all our experiments was a study requirement. It is worth noting that, because of similar issues, other state-of-the-art approaches targeting different game properties were experimented with only one game as well (see *e.g.,* [ZFR14, PLV+20, BGTG20, WCX+20]). We believe that instantiating *RELINE* on a new game would be much easier by collaborating with the game developers. While this would only slightly simplify the definition of a meaningful reward function, the original developers of the game could easily provide through APIs all information needed by *RELINE* (including, *e.g.,* the FPS), cutting away weeks of work.

A.6 Related Work

Three recent studies [PPG21, TdAA21, LZS21] suggest that finding performance issues in video games is a relevant problem, according to both game developers [PPG21, TdAA21] and players [LZS21]. Nevertheless, to the best of our knowledge, no previous work introduced automated approaches for load testing video games. Therefore, in this section, we discuss

some important works on the quality assurance of video games in general. We first introduce the approaches defined in the literature for training agents able to automatically play and win a game. Then, we show how such approaches are used for play-testing for (i) finding functional issues and (ii) assessing game/level design (*e.g.,* finding unbalanced levels).

A.6.1 Training Agents to Play

Reinforcement Learning (RL) is widely used to train agents able to automatically play video games. Mnih *et al.* [MKS+13, MKS+15] presented the first approach based on high-dimensional sensory input (*i.e.,* raw pixels from the game screen) able to automatically learn how to play a game. The authors used a Convolutional Neural Network (CNN) trained with a variant of Q-learning to train their agent. The proposed approach is able to surpass human expert testers in playing some games from the Atari 2600 benchmark.

Vinyals *et al.* [VEB+17] introduced SC2LE, a RL environment based on the game *StarCraft II* that simplifies the development of specialized agents for a multi-agent environment.

Hessel *et al.* [HMVH+18] analyzed six extensions of the DQN algorithm for RL and they reported the combinations that allow to achieve the best results in terms of training time on the Atari 2600 benchmark.

Baker *et al.* [BKM+19] explored the use of RL in a multi-agent environment (*i.e.,* the *hide and seek* game). They report that agents create self-supervised autocurricula [LHLG19], *i.e.,* curricula naturally emerging from competition and cooperation. As a result, the authors found evidence of strategy learning not guided by direct incentives.

Berner *et al.* [AI19] reported that state-of-the-art RL techniques were successfully used in OpenAI Five to train an agent able to play Dota 2 and to defeat the world champion in 2019 (Team OG). Finally, Mesentier *et al.* [dMSLTN17] reported that AI agents could be easily trained to explore the states of a board game (*Ticket to Ride*) performing automated play-testing.

A.6.2 Testing of Video Games

Functional testing of video games aims at finding unexpected behaviors in a game. Defining the test oracle, *i.e.,* determining if a specific game behavior is defective, is not trivial. Several categories of test oracles were identified to determine if a bug was found: *crash* (the game stops working) [PSM17, ZXS+19], *stuck* (the agent can not win the game) [PSM17, ZXS+19], *game balance* (game too easy or too hard) [ZXS+19], *logical* (an invalid state is reached) [ZXS+19], and *user experience bugs* (related to graphic and sound, *e.g.,* glitches) [PSM17, ZXS+19]. While heuristics can be used to find possible crash-, stuck-, and game-balance-related bugs [ZXS+19], logical and user-experience bugs may require the developers to manually define an oracle.

Iftikhar *et al.* [IIKM15] proposed a model-based testing approach for automatically perform black-box testing of platform games. More recent approaches mostly rely on RL.

Pfau *et al.* [PSM17] introduced ICARUS, a framework for autonomous play-testing aimed at finding bugs. ICARUS supports the fully automated detection of *crash* and *stuck* bugs, while it also provides semi-supervised support for *user-experience* bugs.

Zheng *et al.* [ZXS+19] used Deep Reinforcement Learning (DLR) in their approach, Wuji. Wuji balances the aim of winning the game and exploring the space to find *crash*, *stuck*, *game balance*, and *logical* bugs in three video games (one simple, *Block Maze* and two commercial, *L10* and *NSH*).

Bergdahl *et al.* [BGTG20] defined a DLR-based method which provides support for continuous actions (*e.g.,* mouse or game-pads) and they experimented it with a first-person shooter game.

Wu *et al.* [WCX+20] used RL to automatically perform regression testing, *i.e.,* to compare the game behaviors in different versions of a game. They experimented with such an approach on a Massive Multiplayer Online Role-Playing Game (MMORPG).

Ariyurek *et al.* [ABS21] experimented RL and Monte Carlo Tree Search (MCTS) to define both synthetic agents, trained in a completely automated manner, and human-like agents, trained on trajectories used by human testers.

Finally, Ahumada and Bergel [AB20] proposed an approach based on genetic algorithms to reproduce bugs in video games by reconstructing the correct sequence of actions that lead to the desired faulty state of the game.

A.6.3 Game- and Level-Design Assessment

One of the main goals of a video game is to provide a pleasant gameplay to the player. Assessing the game balance and other aspects related to game- and level-design is, therefore, of primary importance.

For this reason, previous work defined several approaches for automatically finding game- and level-design issues in video games. Zook *et al.* [ZFR14] proposed an approach based on Active Learning (AL) to help designers performing low-level parameter tuning. They experimented such an approach on a *shoot 'em up* game.

Gudmundsson *et al.* [GEP+18] introduced an approach based on Deep Learning to learn human-like play-testing from player data. They used a CNN to automatically predict the most natural next action a player would take aiming to estimate difficulty of levels in *Candy Crush Saga* and *Candy Crush Soda Saga*.

Zhao *et al.* [ZBB+19] report four case studies in which they experiment the use of human-like agent trained with RL to predict player interactions with the game and to highlight possible game-design issues. On a similar note, Pfau *et al.* [PLV+20] used deep player behavioral models to represent a specific player population for *Aion*, a MMORPG. They used such models to estimate the game balance and they showed that they can be used to tune it.

Finally, Stahlke *et al.* [SNMB20] defined PathOS, a tool aimed at helping developers to simulate players' interaction with a specific game level, to understand the impact of small design changes.

A.7 Conclusions and Future Work

We presented *RELINE*, an approach that uses RL to load test video games. *RELINE* can be instantiated on different games using different RL models and reward functions.

Our proof-of-concept study performed on two subject systems shows the feasibility of our approach: Given a reward function able to reward the agent when artificial performance bugs are identified, the agent adapts its behavior to play the game while looking for those bugs.

We performed a case study on a real 3D racing game, SuperTuxKart, showing the ability of *RELINE* to identify areas resulting in FPS drops. As compared to a classic RL agent only trained to play the game, *RELINE* is able to identify a substantially higher number of low-FPS points (173 *vs* 33).

Despite the encouraging results, there are many aspects that deserve a deeper investigation and from which our future research agenda stems. First, we plan additional tests on SuperTuxKart to better understand how the agent reacts to changes in the reward function (*e.g.,* is it possible to find more low-FPS points in the central part of the game?). Also, with longer training times it should be possible to train an agent able to play more challenging versions of this game featuring additional 3D effects (*e.g.,* rainy conditions), possibly allowing to find new low-FPS points. We also plan to instantiate *RELINE* on other game genres (*e.g.,* role-playing games), possibly by cooperating with their developers.

A.8 Replication Package

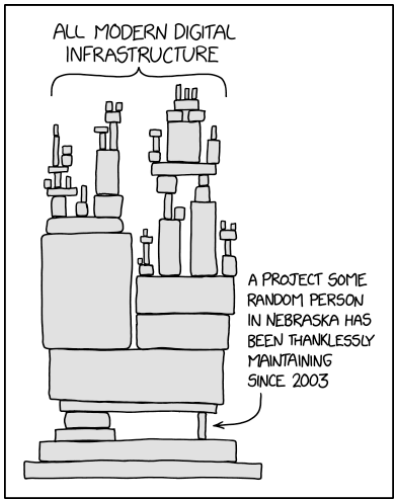
We release all code and data used in our study in a comprehensive replication package [repg]. It contains:

- everything needed to replicate study 1 on CartPole and MsPacman games;
- everything needed to replicate study 2 on SuperTuxKart game;
- all the results obtained and described in this study.

B

Don't Reinvent the Wheel: Towards Automatic Replacement of Custom Implementations with APIs

Reusing code is a common practice in software development: It helps developers speedup the implementation task while also reducing the chances of introducing bugs, given the assumption that the reused code has been tested, possibly in production. Despite these benefits, opportunities for reuse are not always in plain sight and, thus, developers may miss them. We present our preliminary steps in building *RETIWA*, a recommender able to automatically identify custom implementations in a given project that are good candidates to be replaced by open source APIs. *RETIWA* relies on a "knowledge base" consisting of real examples of custom implementation-to-API replacements. In this work, we present the mining strategy we tailored to automatically and reliably extract replacements of custom implementations with APIs from open source projects. This is the first step towards building the envisioned recommender.



The content of this chapter has been presented in the following paper:

> **Don't Reinvent the Wheel: Towards Automatic Replacement of Custom Implementations with APIs**
>
> Rosalia Tufano, Emad Aghajani, Gabriele Bavota. In *Proceedings of the 38th International Conference on Software Maintenance and Evolution (ICSME 2022)*, pp. 394-398.

B.1 Introduction

Code reuse is a well-known practice aimed at improving both developers productivity and code quality [GC92]. There is evidence about the benefits of systematically reusing code, especially for what concerns a lower likelihood of having bugs in reused code [FF96, LGA+07, HvKS08, MC08]. Such benefits arise when the reused code is not outdated [XMYI14] and follows good patterns of code reuse [KG08].

In such a context, Application Programming Interfaces (APIs) provide reusable functionalities that can be exploited by developers to (i) speedup the implementation of new features, and (ii) rely on well-tested implementations that have been possibly deployed in hundreds of client projects. Despite these benefits, developers may not be aware of the availability of a specific feature implementation as an API offered by a third-party library, thus missing the opportunity of reusing it. To reduce such a risk, researchers proposed techniques aimed at recommending APIs given the coding context of the developer (*i.e.,* the code they are currently writing) [HH11, Thu16, NDRDS+21]. While these tools could reduce the risk of reimplementing features offered in well-known libraries, such a scenario is still likely to happen, as also demonstrated by the results we present.

We describe our vision for *RETIWA* (**RE**placing cus**T**om **I**mplementations **W**ith **A**pis), a recommender to identify custom implementations (*i.e.,* code implemented from scratch) that can be replaced by third-party APIs. *RETIWA* is complementary to existing API recommenders [HH11, Thu16, NDRDS+21]: The latter can recommend APIs while the developer is implementing. Differently, *RETIWA* comes into play once the custom code has been already implemented, automatically identifying it as a "clone" of a feature offered by well-known third-party APIs. This implies that *RETIWA* cannot save implementation effort, but can still avoid the maintenance of custom code that can be replaced with well-known and likely well-tested APIs, thus boosting code quality.

We present the overall idea behind *RETIWA* and the first steps we made to build it. We started building a "knowledge base" featuring real replacements of custom implementations with APIs performed by developers in open source projects. We show the challenges behind such a task and the strategies we defined to address it. Such a knowledge base allows to identify common "replacement patterns" that can be used to trigger custom implementation-to-API replacement recommendations.

B.2 Envisioned Tool

Fig. B.1 shows the main steps behind *RETIWA*. First, based on our preliminary tests, it is not realistic to reliably identify commits in which a custom implementation is replaced with an API without having a starting set of known APIs to support. Thus, our approach starts by mining software libraries with the goal of collecting a set of APIs (Step 1 in Fig. B.1) through the *Libraries miner* component. The libraries to support can be defined by the user. The set of mined APIs is then provided as input to the *Client projects analyzer* (Step 2). The latter, given a collection of GitHub repositories, clones and analyzes the change history of the projects to identify candidate replacements where a custom implementation c is replaced with one of

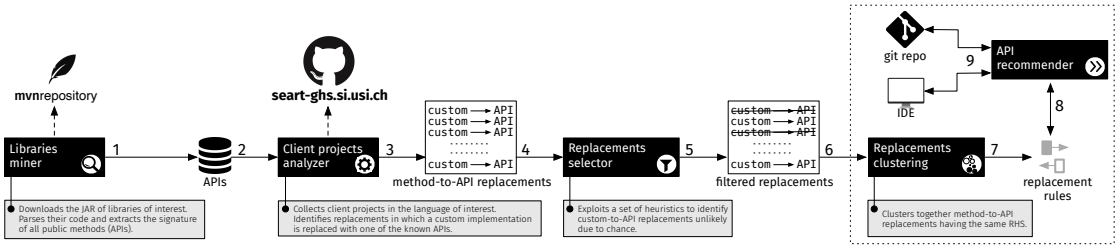


Figure B.1. Overview of *RETIWA* Workflow

the known APIs (Step 3). We indicate these custom implementation to API replacements as $c \rightarrow API$, with c representing the lefthand side (LHS) and API the righthand side (RHS). A single commit may contain multiple replacements, *i.e.,* several custom implementations are replaced with different APIs.

The parsing performed to identify such replacements may result in false positives. For this reason, the *Replacements selector* applies a set of heuristics to exclude from the dataset instances likely to represent false positives (Step 5). The instances surviving such a filter are provided as input to the *Replacements clustering* component (Step 6), which is in charge of grouping replacement instances characterized by the same RHS. These are code changes, possibly coming from different repositories, in which developers replaced a variety of custom implementations with the same API. The output of this step are the actual replacement rules $C \rightarrow API$, with C being a set of custom implementations (Step 7). These rules can be used to recommend to developers the replacement of custom implementations with suitable APIs (Steps 8 and 9). In particular, a clone detector can be used to identify, either in a git repository or directly in the IDE, a component p being similar to one of the clusters of custom implementations (C). Assuming the reliability of such a clone detector, the corresponding RHS (*i.e.,* the associated API) can then be recommended to the developer as a replacement for p. We implemented the steps 1-6 shown in Fig. B.1 by instantiating *RETIWA* to the Java language and by working at method-level granularity: We look for replacement instances in the form $m \rightarrow API$, where m is a Java method.

In Section B.3 we analyze the meaningfulness of the method-to-API replacements that *RETIWA* was able to automatically identify, leaving the finalization of the recommender system (dashed part in Fig. B.1) as future work. In the following, we provide details on the components we implemented and the design decisions we took.

B.2.1 Libraries Miner

The first component is in charge of building a database of potential APIs that our approach may suggest as replacement for an equivalent custom implementation. *Libraries miner* currently supports the collection of APIs from Maven libraries. Given a list of libraries of interest, *Libraries miner* retrieves their source code by downloading and uncompressing the `*-source.jar` for their latest release hosted on the *Maven Central Repository* [mvn22]. The set of public methods (APIs) in each library is then extracted using the *Eclipse Java Parser* [ecl22].

For each identified API we store the following information: package name, the path of the file from which it has been extracted, and its method signature. *Libraries miner* also extracts all `packages` defined in the source code of each library. Such information will be used to identify if `import` statements in client projects refer to any of the target libraries.

B.2.2 Client Projects Analyzer

This component is responsible for cloning a set of given GitHub repositories and analyzing their change history with the goal of identifying potential replacements of a custom method with one of the APIs of interest. The analysis of each client project starts by creating a linear history of commits of the repository's default branch using the `git log <default-branch> --first-parent` command. Then, we iterate through all commits and perform the following steps on every two consecutive snapshots s_i and s_{i+1}, where s_i is the system's snapshot before the changes introduced by commit c_i and s_{i+1} is the snapshot after the changes introduced by c_i.

We start by extracting all *method declarations* and *method invocations* in the s_i and s_{i+1} snapshots. The method declarations represent all internal methods (*i.e.,* methods actually implemented in the system under analysis, excluding those imported from external libraries) existing in the system at a given snapshot. We indicate them with D_{s_i} and $D_{s_{i+1}}$. We use instead I_{s_i} and $I_{s_{i+1}}$ to indicate all method invocations existing in the two snapshots. These lists are extracted by checking out the corresponding snapshot (*i.e.,* `git checkout <commit>`) and parsing the obtained Java files using SrcML [srca]. The main idea behind extracting these lists is to check in the following steps if the commit c_i: (i) replaced all method invocations to an internal method m with invocations to a non-internal API; and (ii) deleted the implementation of the internal method m, replaced by the usage of API.

To perform this check, we start by running the command `git diff` s_i s_{i+1} `--word-diff --unified=0 --ignore-all-space --F.java` for each file $F.java$ modified or renamed in c_i. This version of `git diff` outputs pairs of `[-oldcode-] {+newcode+}` snippets, with the diff algorithm trying to match newly added code fragments (*i.e.,* `newcode`) with deleted code ones (*i.e.,* `oldcode`) when possible. We use such a command to identify invocations to m replaced with invocations to API in commit c_i. This is done by parsing the diff output using SrcML to match replaced method invocations. A few clarifications are needed on this step. First, we only focus on files modified or renamed in c_i since we look for method invocation replacements which cannot happen in added or deleted files. Second, `git diff` relies on heuristics to match deleted and added code fragments when possible, thus being a source of imprecisions in our approach. Third, similarly, SrcML is applied to parse fragments of code in the diff output, thus again resulting in imprecisions when matching method invocations.

To exclude pairs $m \rightarrow API$ which are not of our interest (*e.g., API* is not an actual external API), we make sure that:

(1) D_{s_i} contains m. This ensures that the method m was an internal method declared in snapshot s_i.

(2) The m implementation in s_i is non-empty (*i.e.,* it has a body) and does not invoke any method having the same exact name as API (even with a different signature). This filtering

step ensures that m actually implemented something and, thus, could be a candidate to be replaced with an external API.

Also, it excludes cases in which the developers were already using the API (or a slightly different version of it taking different parameters), with m acting as a wrapper for API. We are not interested in these cases since our goal is to recommend replacements of custom methods with APIs. However, if the developers are already aware of the existence of API, there is no reason for recommending it and probably they had a reason for not using it in the first place.

(3) $I_{s_{i+1}}$ contains API (*i.e.,* the set of invocations in snapshot s_{i+1} must contain the invocation to the candidate API), otherwise SrcML failed to recognize a true method invocation from the `newcode` code fragment.

(4) There is no m declaration in $D_{s_{i+1}}$ nor m invocation in $I_{s_{i+1}}$. This indicates that (i) the implementation of the candidate custom method m has been deleted from the system ($m \notin D_{s_{i+1}}$); and (ii) no invocations to it exist anymore ($m \notin I_{s_{i+1}}$).

(5) There is no method declarations matching API in $D_{s_{i+1}}$, otherwise API is a not an external API method.

The outcome of the aforementioned steps on commit c_i is a set of candidate method-to-API replacements in form of $m \rightarrow API$ for which (i) m is a custom method that has been deleted in c_i; and (ii) API is an external method that has been added in c_i to replace all invocations to m. The *Client projects analyzer* also makes sure that the API is part of the libraries of interest provided as input to the *Libraries miner*. This is done by verifying whether at least one of the added import statements in the files in which a $m \rightarrow API$ replacement happened matches with the list of packages extracted by the *Libraries miner* for the libraries of interest. If this is not the case, the corresponding candidate replacement pair is removed.

The final set of candidate replacement instances for each commit contains: (i) the GitHub repository owner/name; (ii) the commit sha; (iii) custom method information, including its signature and full implementation; (iv) information about the added API, including signature and potential libraries it belongs to based on the `import` statements analysis; (v) file paths in which the replacements occurred; and (vi) the total number invocation replacements occurred in that commit (*i.e.,* the number of times an invocation to m is replaced with API).

B.2.3 Replacements Selector

This component is in charge of further filter out from the set of candidate replacement those instances likely to be false positives. Indeed, while implementing our approach, we noticed that two simple heuristics could be used to remove instances unlikely to be relevant for our goal. First, we exclude all replacement instances in which the custom method m is either a getter, a setter, or a `main` method. We do not see interesting scenarios in which it could make sense to recommend the replacement of these types of methods with APIs.

Second, we conjecture that the number of $m \rightarrow API$ invocation replacements performed in a commit can be an indicator of how "reliable" is the method-to-API replacement we identified. Indeed, if we observe that in a given instance the internal method m is removed and the several invocations to it that were present in the system are replaced with invocations

to *API*, this (i) supports the idea that *m* was a sort of "utility" method invoked in different parts of the code; and (ii) reduces the chances that the replacement is the result of a parsing error in the diff. We study how applying a threshold t on the minimum number of call replacements impacts the reliability of the identified replacement instances (Section B.3).

B.3 Preliminary Study

The *goal* of this study is to assess the feasibility of *RETIWA* in terms of the possibility to automatically identify changes in which custom implementations are replaced with APIs.We aim at answering $\mathbf{RQ}_1$: *To what extent is it possible to automatically identify method-to-API replacements?* The automatic identification of $m \rightarrow API$ replacements is a pre-requisite for building *RETIWA*. RQ_1 looks at the precision of the approaches described in Section B.2.

B.3.1 Study Design

The *Libraries miner* collected 38 Apache `commons` libraries by leveraging the MVN repository website [mvn22]. These libraries can be identified as those having `org.apache.commons` as maven group-id and have been downloaded and parsed as described in Section B.2.1. We used the SEART GitHub Search [ghs] to collect as client projects all non-fork Java GitHub repositories having at least 500 commits and 10 stars. These filters have been set in an attempt to exclude toy/personal projects. The *Client projects analyzer* obtained 9,788 repositories as result of this search. However, only a fraction of these repositories (1,856) declared a dependency towards one of the considered libraries during their change history. These are the only projects from which we can expect useful data points for our study.

The analysis of these repositories, performed as described in Section B.2, resulted in 337 candidate replacements which we manually analyzed to answer RQ_1. In particular, each $m \rightarrow API$ candidate replacement was inspected independently by two researchers with the goal of classifying it as a true or false positive. To come up with such a classification, the inspector looked at (i) the diff of the commit on GitHub, (ii) a summary we created featuring all the invocations to *m* that were replaced with an invocation to *API*, and (iii) the commit note. Conflicts, that arisen for 34 instances (10%), have been solved by a third researcher not involved in the original classification. We report the precision of the identified instances (*i.e.,* the percentage of true positives among the candidate replacements) and discuss how it can be improved by increasing the minimum number t of *call replacements* (see Section B.2.3).

B.3.2 Results Discussion

Table B.1 reports the results achieved as output of our manual validation. The first column shows the threshold applied as additional filtering criterion to remove replacement commits not featuring at least t call replacements.

The first row (*i.e.,* $t \geq 1$) represents the scenario in which such a filter is not applied, since all instances *RETIWA* identified will have by construction at least one call replacement.

Table B.1. Manual analysis of 337 replacements identified by *RETIWA*

t	# Instances	# True Positives	Precision (%)
≥ 1	337	165	48.9%
≥ 2	80	67	83.8%
≥ 3	46	39	84.8%
≥ 4	33	28	84.8%
≥ 5	25	23	92.0%

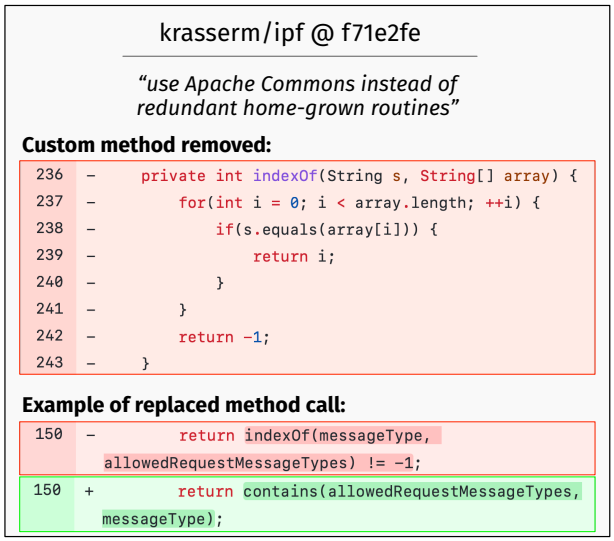


Figure B.2. Replacing the custom `indexOf` method with `contains` API

As it can be seen, without any additional filtering the precision of the 337 identified replacements is limited to 48.9%. By increasing the value of the t threshold, the precision quickly increases, with a 83.8% already achieved with $t \geq 2$ (*i.e.,* at least two invocations to the custom method m have been replaced with the *API* in the commit). Clearly, such an increase in precision has a cost in terms of true positive replacements that are excluded as having $t < 2$ (165-67=98 true positives are excluded). This is the usual recall *vs* precision tradeoff that should be assessed based on how the envisioned recommender system will be built on top of this data. One option is also to provide developers with the possibility to decide the value of t: Higher values will result in less recommendations likely to be of high quality, while lower values, especially 1, will trigger more recommendations including, however, a higher percentage of false positives. In the following we discuss three concrete examples of replacements we found, while the whole dataset is available in our replication package [repb].

In Fig. B.2 the name of the GitHub repository and the commit we refer to are shown at the top. Following in *italic* is the commit message used by the developer who explicitly indicates the aim of the commit of removing "*home-grown routines*" in favor of APIs implemented in Apache commons. In the reported example the custom method `indexOf` returns the index

of a given `String` in the `array` provided as parameter or `-1` if the `array` does not contain the `String`.

This method is used in the system to check for the existence of the given `String` in the `array` as it can be seen from the replaced method call, in which it is used to return `true` in case `indexOf` returns a value `!= -1`. The replacing API `contains` already returns a `boolean` thus also simplifying the locations in the code in which it is used instead of `indexOf`.

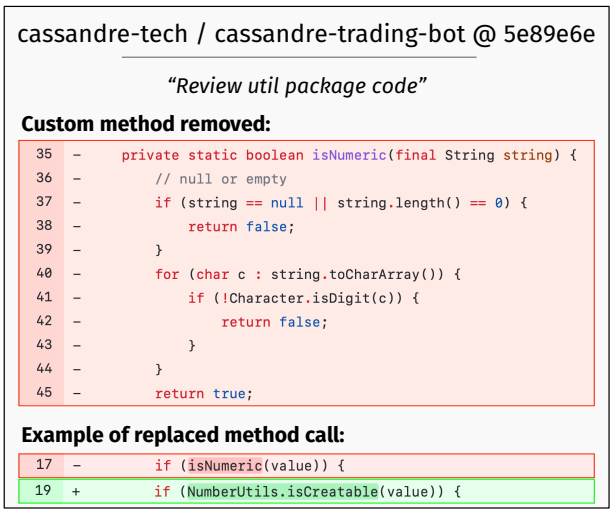


Figure B.3. Replacing the custom `isNumeric` method with `isCreatable` API

Fig. B.3 reports another example of replacement *RETIWA* identified. Differently from the previous commit, the commit message in this case does not allow to infer the presence of a $m \rightarrow API$ replacement without looking at the code diff. This is something we observed in the vast majority of commits in our dataset. Indeed, our initial idea was to match textual patterns in the commit messages to identify the candidate replacement commits as those containing *e.g.,* "*replaced custom [*] with [*]*". However, such an approach is simply not an option due to the limited number of commits explicitly documenting these changes in their message.

In this example, the replaced custom method is `isNumeric`, which was in charge of verifying whether a `String` provided as parameter was composed by only numbers. Invocations to such a method have been replaced by the `NumerUtils.isCreatable` API which also takes as input a `String` and, accordingly to its documentation, "*checks whether the String is a valid Java number*".

Finally, Fig. B.4 depicts a commit, again characterized by a rather vague "*various improvements*" message, in which the custom `randomString` method used to generate random strings has been replaced with the `randomAlphabetic` Apache API.

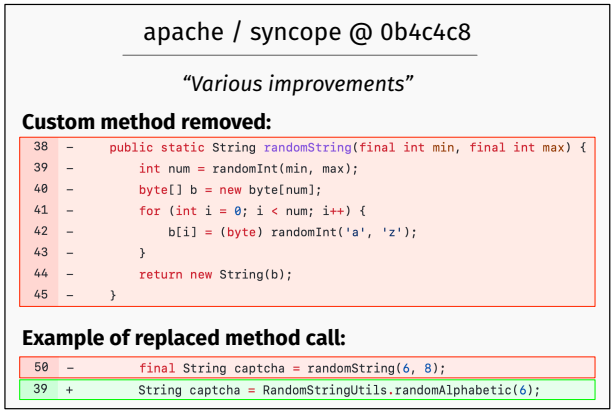


Figure B.4. Replacing the custom `randomString` method with `randomAlphabetic` API

Answer to RQ$_1$. The automatic identification of changes replacing custom implementations with APIs is challenging but feasible. Indeed, the approach we presented was able to identify 337 of these commits with a precision of 48.9% and, using specific filtering heuristics (*e.g.,* $t \geq 2$), the precision level can be substantially boosted to $> 80\%$. More research is needed to optimize the recall *vs* precision tradeoff.

B.3.3 Threats to Validity

Construct validity. Our parsing procedure exploiting the output of the `git diff` and SrcML may be subject to imprecisions when identifying the custom method invocations, the API invocations, and when mapping the API to the corresponding library through the analysis of the `import` statements. Still, our goal was to investigate the feasibility of our approach and we are aware that better implementations based on full static code analysis of the involved snapshots can increase the parsing accuracy.

Internal validity. Subjectiveness in the manual analyses could have affected our results. To mitigate such a bias, when classifying the candidate replacement commits as *true* or *false positives*, two researchers independently classified each comment, and a third one was involved in case of conflict. Despite such a process, imprecisions are still possible. The output of our manual analysis is publicly available for inspection [repb].

External validity. Our preliminary study focuses on a set of 38 Apache libraries and 1,856 Java client projects. Most of the steps behind *RETIWA* are independent from the target language, assuming the reimplementation of the low-level components such as the parser. Larger studies involving more diverse set of libraries are planned to corroborate our findings and designing the final version of *RETIWA*.

B.4 Conclusions and Future Work

We presented our vision for *RETIWA*, an approach aimed at automatically identifying custom implementations that can be replaced by well-known third-party APIs. *RETIWA* takes as input a set of libraries of interest that are parsed to identify the APIs they contain. Then, a large set of client projects is mined to identify code changes in which developers replace a custom implementation with one of the known APIs. Heuristics are then applied to filter out false positives. By using the process we propose it is possible to automatically identify these replacements with a 48.9% precision and such a precision can be boosted to >80% by increasing the threshold t (see Section B.2.3). The replacement changes represent the basic data on top of which we plan to build the full recommender system depicted in Fig. B.1.

Towards this goal, our future work will span three main directions. First, we plan to explore alternatives for a more reliable code parsing. The obvious one is to perform a full parsing of the snapshots before and after each commit to identify custom code replaced with APIs. Such a process, while precise, is extremely expensive when applied on thousands of systems (*i.e.,* hundreds of thousands of commits to analyze).

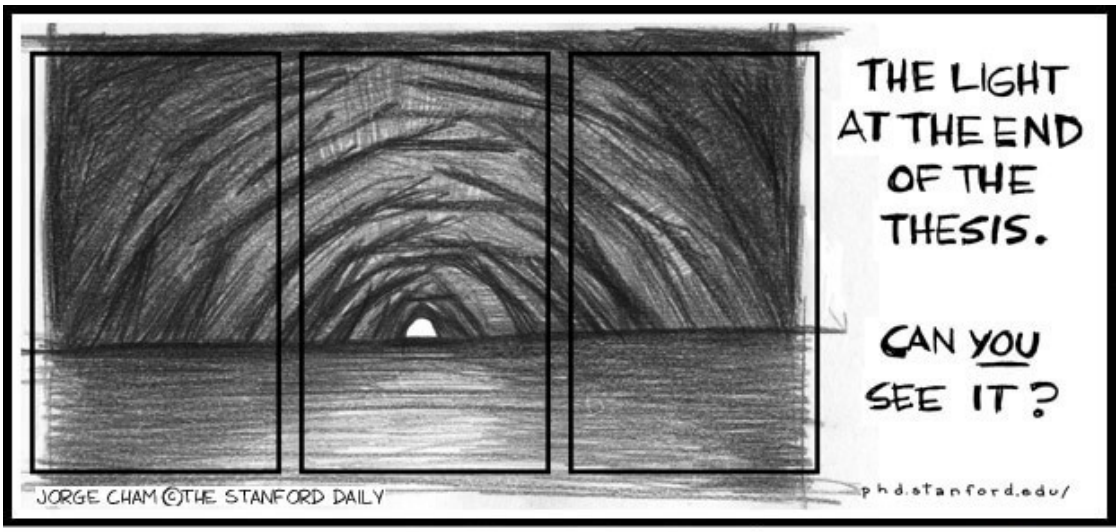
Second, the combination of heuristics we adopt to identify replacement commits may be suboptimal and exclude several true positives. For example, *RETIWA* only selects as candidate replacement commits those in which a known `import` statement (*i.e.,* an `import` statement coming from one of the parsed libraries) is added in the commit. This is based on the assumption that the API usage implies the addition of the `import` statement.

Third, once the identification of replacements is crystallized, we will focus on implementing the two remaining steps of *RETIWA*, namely the clustering and the triggering of recommendations, as described in Section B.2.

B.5 Replication Package

We release all code and data used in our study in a comprehensive replication package [repb]. It contains:

- the code for the first three *RETIWA* components: *Libraries Miner*, *API Extractor* and *Client Project Analyzer*;
- the list of 38 libraries parsed for creating API knowledge based;
- the information of the 1033 $m \rightarrow API$ replacements found in 10k parsed client projects;
- the result of the manual analysis on the data.


THE LIGHT AT THE END OF THE THESIS.
CAN YOU SEE IT?
JORGE CHAM ©THE STANFORD DAILY
phd.stanford.edu/

Bibliography

[3dc] 3d.city - performance issue 42. https://github.com/lo-th/3d.city/issues/42.

[AB20] Tomás Ahumada and Alexandre Bergel. Reproducing bugs in video games using genetic algorithms. In *2020 IEEE Games, Multimedia, Animation and Multiple Realities Conference (GMAX)*, pages 1–6, 2020.

[ABIR17] Toufique Ahmed, Amiangshu Bosu, Anindya Iqbal, and Shahram Rahimi. Senticr: a customized sentiment analysis tool for code review interactions. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 106–111. IEEE, 2017.

[ABS21] S. Ariyurek, A. Betin-Can, and E. Surer. Automated video game testing using synthetic and humanlike agents. *IEEE Transactions on Games*, 13(1):50–67, 2021.

[ACM] Acm digital library. https://dl.acm.org/. Accessed: 2022-11-15.

[ACO+20] Krishna Teja Ayinala, Kwok Sun Cheng, Kwangsung Oh, Teukseob Song, and Myoungkyu Song. Code inspection support for recurring changes with deep learning in evolving software. In *2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 931–942, 2020.

[ACRC21] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation. In *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT*, pages 2655–2668, 2021.

[ADS22] Toufique Ahmed, Premkumar T. Devanbu, and Anand Ashok Sawant. Learning to find usages of library functions in optimized binaries. *IEEE Trans. Software Eng.*, 48(10):3862–3876, 2022.

[AI19] Open AI. Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680*, 2019.

[AKB+19] Sumit Asthana, Rahul Kumar, Ranjita Bhagwan, Christian Bird, Chetan Bansal, Chandra Shekhar Maddila, Sonu Mehta, and Balasubramanyan Ashok. Whodo: Automating reviewer suggestions at scale. In *27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE*, page 937–945, 2019.

[APS16] Miltiadis Allamanis, Hao Peng, and Charles Sutton. A convolutional attention network for extreme summarization of source code. In *33nd International Conference on Machine Learning, ICML*, pages 2091–2100, 2016.

[ATD+20] Wisam Haitham Abbood Al-Zubaidi, Patanamon Thongtanunam, Hoa Khanh Dam, Chakkrit Tantithamthavorn, and Aditya Ghose. Workload-aware reviewer recommendation using a multi-objective search-based approach. In *16th ACM International Conference on Predictive Models and Data Analytics in Software Engineering, PROMISE*, pages 21–30, 2020.

[ATLJ20] Muhammad Hilmi Asyrofi, Ferdian Thung, David Lo, and Lingxiao Jiang. Crossasr: Efficient differential testing of automatic speech recognition via text-to-speech. In *36th IEEE International Conference on Software Maintenance and Evolution ICSME*, pages 640–650, 2020.

[AWEA23] Ishan Aryendu, Ying Wang, Farah Elkourdi, and Eman Abdullah Alomar. Intelligent code review assignment for large scale open source software stacks. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, ASE*, page to appear, 2023.

[AZFL22] Waad Alhoshan, Liping Zhao, Alessio Ferrari, and Keletso J. Letsholo. A zero-shot learning approach to classifying requirements: A preliminary study. In *Requirements Engineering: Foundation for Software Quality: 28th International Working Conference, REFSQ*, pages 52–59, 2022.

[AZLY19] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. Code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL):40:1–40:29, 2019.

[Bal13] Vipin Balachandran. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 931–940. IEEE, 2013.

[BB13] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *35th IEEE/ACM International Conference on Software Engineering, ICSE*, pages 712–721, 2013.

[BBaBL15] Mike Barnett, Christian Bird, Jo ao Brunet, and Shuvendu K. Lahiri. Helping developers help themselves: Automatic decomposition of code review changesets. In *37th IEEE/ACM International Conference on Software Engineering, ICSE*, pages 134–144, 2015.

[BBBK11] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *25th Advances in Neural Information Processing Systems NIPS*, pages 2546–2554, 2011.

[BBV13] Nicolas Boulanger-Lewandowski, Yoshua Bengio, and Pascal Vincent. Audio chord recognition with recurrent neural networks. In *14th International Society for Music Information Retrieval Conference, ISMIR*, pages 335–340, 2013.

[BBZJ14] Moritz Beller, Alberto Bacchelli, Andy Zaidman, and Elmar Juergens. Modern code reviews in open-source projects: Which problems do they fix? In *11th IEEE/ACM Working Conference on Mining Software Repositories, MSR*, pages 202–211, 2014.

[BC96] Fevzi Belli and Radu Crisan. Towards automation of checklist-based code-reviews. In *Proceedings of ISSRE'96: 7th International Symposium on Software Reliability Engineering*, pages 24–33. IEEE, 1996.

[BC13] A. Bosu and J. C. Carver. Impact of peer code review on peer impression formation: A survey. In *7th IEEE/ACM International Symposium on Empirical Software Engineering and Measurement, ESEM*, pages 133–142, 2013.

[BCB15] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *3rd International Conference on Learning Representations, ICLR*, 2015.

[BCP+16] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[BGTG20] J. Bergdahl, C. Gordillo, K. Tollmar, and L. Gisslén. Augmenting automated game testing with deep reinforcement learning. In *2020 IEEE Conference on Games (CoG)*, pages 600–603, 2020.

[BHRV21] Berkay Berabi, Jingxuan He, Veselin Raychev, and Martin Vechev. Tfix: Learning to fix coding errors with a text-to-text transformer. In *38th International Conference on Machine Learning, ICML*, pages 780–791, 2021.

[BJK06] Bum Hyun Lim, Jin Ryong Kim, and Kwang Hyun Shim. A load testing architecture for networked virtual environment. In *2006 8th International Conference Advanced Communication Technology*, volume 1, pages 5 pp.–848, 2006.

[BKM+19] Bowen Baker, Ingmar Kanitscheider, Todor Markov, Yi Wu, Glenn Powell, Bob McGrew, and Igor Mordatch. Emergent tool use from multi-agent autocurricula. *arXiv preprint arXiv:1909.07528*, 2019.

[BKS18] Sahil Bhatia, Pushmeet Kohli, and Rishabh Singh. Neuro-symbolic program corrector for introductory programming assignments. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 60–70. IEEE, 2018.

[BR15] Gabriele Bavota and Barbara Russo. Four eyes are better than two: On the impact of code reviews on software quality. In *31th IEEE International Conference on Software Maintenance and Evolution, ICSME*, pages 81–90, 2015.

[Bre01] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[BV21] Rodrigo Brito and Marco Tulio Valente. Raid: Tool support for refactoring-aware code reviews. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pages 265–275. IEEE, 2021.

[BYC13a] James Bergstra, Daniel Yamins, and David Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *International conference on machine learning*, pages 115–123, 2013.

[BYC+13b] James Bergstra, Daniel Yamins, David Cox, D David, et al. Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms. In *12th Python in science conference, SciPy*, page 20, 2013.

[Car] Cartpole. `https://gym.openai.com/envs/CartPole-v0/`.

[CBHK02] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research, JAIR*, 16(1):321–357, 2002.

[CCP+21a] 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, TSE*, abs/2108.01585(01):1–1, 2021.

[CCP+21b] Matteo Ciniselli, Nathan Cooper, Luca Pascarella, Denys Poshyvanyk, Massimiliano Di Penta, and Gabriele Bavota. An empirical study on the usage of BERT models for code completion. In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021*, pages 108–119, 2021.

[CGT15] Jacek Czerwonka, Michaela Greiler, and Jack Tilford. Code reviews do not find bugs: How the current code review best practice slows us down. In *37th IEEE/ACM International Conference on Software Engineering, ICSE*, pages 27–28, 2015.

[cha] Chatgpt. `https://openai.com/blog/chatgpt`. Accessed: 2023-03-27.

[CKT+21] 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 Trans. Software Eng.*, 47(9):1943–1959, 2021.

[CLBZ20] Aleksandr Chueshev, Julia Lawall, Reda Bendraou, and Tewfik Ziadi. Expanding the number of reviewers in open-source projects by recommending appropriate developers. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 499–510, 2020.

[CLS+10] C. Cho, D. Lee, K. Sohn, C. Park, and J. Kang. Scenario-based approach for blackbox load testing of online game servers. In *2010 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery*, pages 259–265, 2010.

[CMKS17] Zhiyuan Chen, Maneesh Mohanavilasam, Young-Woo Kwon, and Myoungkyu Song. Tool support for managing clone refactorings to facilitate code review in evolving software. In *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 288–297. IEEE, 2017.

[COM+21] Moataz Chouchen, Ali Ouni, Mohamed Wiem Mkaouer, Raula Gaikovina Kula, and Katsuro Inoue. Whoreview: A multi-objective search-based approach for code reviewers recommendation in modern code review. *Applied Soft Computing*, 100:106908, 2021.

[Con98] W. J. Conover. *Practical Nonparametric Statistics*. Wiley, 3rd edition edition, 1998.

[COOM23] Moataz Chouchen, Ali Ouni, Jefferson Olongo, and Mohamed Wiem Mkaouer. Learning to predict code review completion time in modern code review. *Empirical Software Engineering*, 2023.

[Cox99] A.P.M. Coxon. *Sorting Data: Collection and Analysis*. Number no. 127 in Quantitative Applications in the Social Sciences. 1999.

[CPML18] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. Compiler fuzzing through deep learning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 95–105, 2018.

[CR21] Saikat Chakraborty and Baishakhi Ray. On multi-modal learning of editing source code. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE*, pages 443–455, 2021.

[CRN22] Lawrence Chen, Peter C. Rigby, and Nachiappan Nagappan. Understanding why we cannot model how long a code review will take: An industrial case study. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE*, page 1314–1319, 2022.

[CSM+18] C. Chen, T. Su, G. Meng, Z. Xing, and Y. Liu. From ui design image to gui skeleton: A neural machine translator to bootstrap mobile gui implementation. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 665–676, 2018.

[DAB21] Ozren Dabic, Emad Aghajani, and Gabriele Bavota. Sampling projects in github for MSR studies. In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR*, pages 560–564, 2021.

[Dev] DEVINTA, an artificial assistant for software developers. http://devinta.si.usi.ch/.

[dl4] https://seart-dl4se.si.usi.ch/.

[DMP+17] J. Deshmukh, A. K. M, S. Podder, S. Sengupta, and N. Dubash. Towards accurate duplicate bug retrieval using deep learning techniques. In *33th IEEE International Conference on Software Maintenance and Evolution ICSME*, pages 115–124, 2017.

[dMSLTN17] Fernando de Mesentier Silva, Scott Lee, Julian Togelius, and Andy Nealen. Ai as evaluator: Search driven playtesting of modern board games. In *AAAI Workshops*, 2017.

[DPSSMB21] Pedro Delgado-Pérez, Ana Belén Sánchez, Sergio Segura, and Inmaculada Medina-Bulo. Performance mutation testing. *Software Testing, Verification and Reliability*, 31(5), 2021.

[dwaa] Dwarfcorp - performance issue 583. https://github.com/Blecki/dwarfcorp/issues/583.

[dwab] Dwarfcorp - performance issue 64. https://github.com/Blecki/dwarfcorp/issues/64.

[dwac] Dwarfcorp - performance issue 904. https://github.com/Blecki/dwarfcorp/issues/904.

[dwad] Dwarfcorp - performance issue 966. https://github.com/Blecki/dwarfcorp/issues/966.

[DWSS21] Dawn Drain, Chen Wu, Alexey Svyatkovskiy, and Neel Sundaresan. Generating bug-fixes using pretrained transformers. In *5th ACM/SIGPLAN International Symposium on Machine Programming @ PLDI, MAPS*, pages 1–8, 2021.

[ecl22] Eclipse JDT Core. https://www.eclipse.org/jdt/core/, 2022.

[Els] Elsevier sciencedirect. https://www.sciencedirect.com/. Accessed: 2022-11-15.

[EMHK+20] Carolyn D Egelman, Emerson Murphy-Hill, Elizabeth Kammer, Margaret Morrow Hodges, Collin Green, Ciera Jaspan, and James Lin. Predicting developers' negative feelings about code review. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 174–185, 2020.

[eng] English stopwords. https://code.google.com/p/stop-words/. Accessed: 2022-11-10.

[FA11] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *21st ACM Joint Meeting of the European Software Engineering Conference and the ACM/SIGSOFT Symposium on the Foundations of Software Engineering, ESEC-FSE*, pages 416–419, 2011.

[FF96] W.B. Frakes and C.J. Fox. Quality improvement using a software reuse failure modes model. *IEEE Transactions on Software Engineering*, 22(4):274–279, 1996.

[FFSB23] Enrico Fregnan, Josua Fröhlich, Davide Spadini, and Alberto Bacchelli. Graph-based visualization of merge requests for code review. *Journal of Systems and Software*, 195:111506, 2023.

[FGT+20] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages. In *10th Conference on Empirical Methods in Natural Language Processing, EMNLP*, pages 1536–1547, 2020.

[FMB+14] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *29th IEEE/ACM International Conference on Automated Software Engineering, ASE*, pages 313–324, 2014.

[FPS18] Mikołaj Fejzer, Piotr Przymus, and Krzysztof Stencel. Profile based recommendation of code reviewers. *Journal of Intelligent Information Systems*, 50:597–619, 2018.

[FS21] Muntazir Fadhel and Emil Sekerinski. Striffs: Architectural component diagrams for code reviews. In *2021 International Conference on Code Quality (ICCQ)*, pages 69–78. IEEE, 2021.

[fun] FunCom dataset. `http://leclair.tech/data/funcom/`.

[FXLL18] Yuanrui Fan, Xin Xia, David Lo, and Shanping Li. Early prediction of merged code changes to prioritize reviewing tasks. *Empirical Software Engineering*, 23:3346–3393, 2018.

[GC92] J.E. Gaffney and R.D. Cruickshank. A general economics model of software reuse. In *International Conference on Software Engineering*, pages 327–337, 1992.

[geo] Geostrike - performance issue 214. `https://github.com/Webiks/GeoStrike/issues/214`.

[GEP+18] Stefan Freyr Gudmundsson, Philipp Eisen, Erik Poromaa, Alex Nodet, Sami Purmonen, Bartlomiej Kozakowski, Richard Meurling, and Lele Cao. Human-like playtesting with deep learning. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8, 2018.

[ger] Gerrit code review. https://www.gerritcodereview.com/. Accessed: 2022-11-10.

[ghs] Msr mining platform. https://seart-ghs.si.usi.ch.

[git] Github. https://github.com/. Accessed: 2022-11-10.

[GJMM21] Lobna Ghadhab, Ilyes Jenhani, Mohamed Wiem Mkaouer, and Montassar Ben Messaoud. Augmenting commit classification by using fine-grained source code changes and a pre-trained deep neural language model. *Inf. Softw. Technol.*, 135:106566, 2021.

[GK05a] Robert J Grissom and John J Kim. *Effect sizes for research: A broad practical approach*. 2005.

[GK05b] Robert J. Grissom and John J. Kim. *Effect sizes for research: A broad practical approach*. Lawrence Earlbaum Associates, 2nd edition edition, 2005.

[GPS17] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&fuzz: machine learning for input fuzzing. In *32nd IEEE/ACM International Conference on Automated Software Engineering, ASE*, pages 50–59, 2017.

[Gra06] E.W. Grafarend. *Linear and Nonlinear Models: Fixed Effects, Random Effects, and Mixed Models*. Walter de Gruyter, 2006.

[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, Virtual Event, Austria, May 3-7, 2021*, 2021.

[GS18] Anshul Gupta and Neel Sundaresan. Intelligent code reviews using deep learning. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'18) Deep Learning Day*, 2018.

[GXL+21] Zhipeng Gao, Xin Xia, David Lo, John Grundy, and Thomas Zimmermann. Automating the removal of obsolete todo comments. In *29th ACM Joint European Software Engineering Conference and the ACM/SIGSOFT International Symposium on the Foundations of Software Engineering ESEC-FSE*, pages 218–229, 2021.

[GYH+20] Chenkai Guo, Hui Yang, Dengrong Huang, Jianwen Zhang, Naipeng Dong, Jing Xu, and Jingwen Zhu. Review sharing via deep semi-supervised code clone detection. *IEEE Access*, 8:24948–24965, 2020.

[Gym] Gym. https://gym.openai.com/.

[GZZK16] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. Deep api learning. In *24th ACM/SIGSOFT International Symposium on Foundations of Software Engineering FSE*, pages 631–642, 2016.

[HCC+21] Haytham Hijazi, José Cruz, João Castelhano, Ricardo Couceiro, Miguel Castelo-Branco, Paulo de Carvalho, and Henrique Madeira. ireview: an intelligent code review evaluation tool using biofeedback. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*, pages 476–485. IEEE, 2021.

[HD17] Vincent J. Hellendoorn and Premkumar Devanbu. Are deep neural networks the best choice for modeling source code? In *11th ACM/SIGSOFT Joint Meeting on Foundations of Software Engineering ESEC-FSE*, page 763?773, 2017.

[HDC+22] Haytham Hijazi, Joao Duraes, Ricardo Couceiro, Joao Castelhano, Raul Barbosa, Júlio Medeiros, Miguel Castelo-Branco, Paulo De Carvalho, and Henrique Madeira. Quality evaluation of modern code reviews through intelligent biometric program comprehension. *IEEE Transactions on Software Engineering*, (01):1–1, 2022.

[HH11] Lars Heinemann and Benjamin Hummel. Recommending api methods based on identifier contexts. In *Proceedings of the 3rd International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation*, SUITE '11, page 1?4, 2011.

[HHH+22] Faria Huq, Masum Hasan, Md Mahim Anjum Haque, Sazan Mahbub, Anindya Iqbal, and Toufique Ahmed. Review4repair: Code review aided automatic program repairing. *Information and Software Technology*, 143:106765, 2022.

[HHJC21] Geert Heyman, Rafael Huysegems, Pascal Justen, and Tom Van Cutsem. Natural language-guided programming. In *Onward! 2021: Proceedings of the 2021 ACM/SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, OOPSLA*, pages 39–55, 2021.

[HHL11] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *5th International Conference on Learning and Intelligent Optimization, LION*, pages 507–523, 2011.

[HII+21] Masum Hasan, Anindya Iqbal, Mohammad Rafid Ul Islam, AJM Imtiajur Rahman, and Amiangshu Bosu. Using a balanced scorecard to identify opportunities to improve code review effectiveness: An industrial experience report. *Empirical Software Engineering*, 26:1–34, 2021.

[HJC+20] Yuan Huang, Nan Jia, Xiangping Chen, Kai Hong, and Zibin Zheng. Code review knowledge perception: Fusing multi-features for salient-class location. *IEEE Transactions on Software Engineering*, 48(5):1463–1479, 2020.

[HMBI17a] Gali Halevi, Henk Moed, and Judit Bar-Ilan. Suitability of google scholar as a source of scientific information and as a source of data for scientific evaluation: Review of the literature. *Journal of Informetrics*, 11(3):823–834, 2017.

[HMBI17b] Gali Halevi, Henk Moed, and Judit Bar-Ilan. Suitability of google scholar as a source of scientific information and as a source of data for scientific evaluation: Review of the literature. *Journal of Informetrics, JOI*, 11(3):823–834, 2017.

[HMS20] Pinjia He, Clara Meister, and Zhendong Su. Structure-invariant testing for machine translation. In *42nd IEEE/ACM International Conference on Software Engineering, ICSE*, pages 961–973, 2020.

[HMVH+18] Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[Hol79a] S. Holm. A simple sequentially rejective bonferroni test procedure. *Scandinavian Journal on Statistics*, 6(2):65–70, 1979.

[Hol79b] Sture Holm. A simple sequentially rejective multiple test procedure. *Scandinavian journal of statistics*, pages 65–70, 1979.

[HOL+18] Jacob A. Harer, Onur Ozdemir, Tomo Lazovich, Christopher P. Reale, Rebecca L. Russell, Louis Y. Kim, and Peter Chin. Learning to repair software vulnerabilities with generative adversarial networks. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, page 7944–7954. Curran Associates Inc., 2018.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation, NECO*, 9(8):1735–1780, 1997.

[HST+21] Jordan Henkel, Denini Silva, Leopoldo Teixeira, Marcelo d'Amorim, and Thomas Reps. Shipwright: A human-in-the-loop system for dockerfile repair. In *43th IEEE/ACM International Conference on Software Engineering, ICSE*, pages 1148–1160, 2021.

[HTT22] Yang Hong, Chakkrit Kla Tantithamthavorn, and Patanamon Pick Thongtanunam. Where should i look at? recommending lines that reviewers should pay attention to. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 1034–1045. IEEE, 2022.

[HTTA22] Yang Hong, Chakkrit Tantithamthavorn, Patanamon Thongtanunam, and Aldeida Aleti. Commentfinder: A simpler, faster, more accurate code review comments recommendation. In *30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC-FSE*, page 507–519, 2022.

[HvKS08] Stefan Haefliger, Georg von Krogh, and Sebastian Spaeth. Code reuse in open source software. *Management Science*, 54(1):180–193, 2008.

[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.

[HWZ13] Jiantao He, Linzhang Wang, and Jianhua Zhao. Supporting automatic code review via design. In *2013 IEEE Seventh International Conference on Software Security and Reliability Companion*, pages 211–218. IEEE, 2013.

[hyp] Hyperopt. `https://github.com/hyperopt/hyperopt/`. Accessed: 2022-11-10.

[HZ13] Kim Herzig and Andreas Zeller. The impact of tangled code changes. In *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13*, pages 121–130, 2013.

[IAS+22] Khairul Islam, Toufique Ahmed, Rifat Shahriyar, Anindya Iqbal, and Gias Uddin. Early prediction for merged vs abandoned code changes in modern code reviews. *Information and Software Technology*, 142:106756, 2022.

[IEE] Ieee xplore digital library. `https://ieeexplore.ieee.org/`. Accessed: 2022-11-15.

[IIKM15] S. Iftikhar, M. Z. Iqbal, M. U. Khan, and W. Mahmood. An automated model based testing approach for platform games. In *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 426–435, 2015.

[java] Java parser. `https://javaparser.org`. Accessed: 2022-11-28.

[javb] javalanche. `https://www.st.cs.uni-saarland.de/mutation/`. Accessed: 2022-11-28.

[Jes] Jester. `http://jester.sourceforge.net`. Accessed: 2022-11-28.

[JLS+05] YungWoo Jung, Bum-Hyun Lim, Kwang-Hyun Sim, HunJoo Lee, IlKyu Park, JaeYong Chung, and Jihong Lee. Venus: The online game simulator using massively virtual clients. In *Systems Modeling and Simulation: Theory and Applications*, pages 589–596, 2005.

[JLT21] Nan Jiang, Thibaud Lutellier, and Lin Tan. Cure: Code-aware neural machine translation for automatic program repair. In *43th IEEE/ACM International Conference on Software Engineering, ICSE*, pages 1161–1173, 2021.

[JLZ+19] Jing Jiang, David Lo, Jiateng Zheng, Xin Xia, Yun Yang, and Li Zhang. Who should make decision on this pull request? analyzing time-decaying relationships and file similarities for integrator prediction. *J. Syst. Softw.*, 154(C):196–210, 2019.

[JYH+17] Jing Jiang, Yun Yang, Jiahuan He, Xavier Blanc, and Li Zhang. Who should comment on this pull request? analyzing attributes for more accurate commenter recommendation in pull-based development. *Inf. Softw. Technol.*, 84(C):48–62, apr 2017.

[KC07] Barbara Kitchenham and Stuart Charters. Guidelines for performing systematic literature reviews in software engineering. 2007.

[KG08] Cory Kapser and Michael W. Godfrey. "cloning considered harmful" considered harmful: patterns of cloning in software. *Empir. Softw. Eng.*, 13(6):645–692, 2008.

[KKD+18] Guillaume Klein, Yoon Kim, Yuntian Deng, Vincent Nguyen, Jean Senellart, and Alexander M. Rush. Opennmt: Neural machine translation toolkit. In *13th Conference of the Association for Machine Translation in the Americas, AMTA*, pages 177–184, 2018.

[KR18] Taku Kudo and John Richardson. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In *8th Conference on Empirical Methods in Natural Language Processing, EMNLP*, pages 66–71, 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.

[lana] Language detection library ported from google's language-detection. `https://pypi.org/project/langdetect/`. Accessed: 2023-10-10.

[lanb] Language identification tool. `https://github.com/saffsd/langid.py`. Accessed: 2022-11-28.

[Lap18] Maxim Lapan. *Deep Reinforcement Learning Hands-On: Apply Modern RL Methods, with Deep Q-Networks, Value Iteration, Policy Gradients, TRPO, AlphaGo Zero and More*. Packt Publishing, 2018.

[LBH16] Dayi Lin, C. Bezemer, and A. Hassan. Studying the urgent updates of popular games on the steam platform. *Empirical Software Engineering*, 22:2095–2126, 2016.

[Lev66] VI Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10(8):707, 1966.

[LGA+07] Jingyue Li, Anita Gupta, Jon Arvid, Borretzen Borretzen, and Reidar Conradi. The empirical studies on quality benefits of reusing software components. In *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, volume 2, pages 399–402, 2007.

[LHL+17] Sun-Ro Lee, Min-Jae Heo, Chan-Gun Lee, Milhan Kim, and Gaeul Jeong. Applying deep learning based automatic bug triager to industrial projects. In *11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE*, pages 926–931, 2017.

[LHLG19] Joel Z Leibo, Edward Hughes, Marc Lanctot, and Thore Graepel. Autocurricula and the emergence of innovation from social interaction: A manifesto for multi-agent intelligence research. *arXiv preprint arXiv:1903.00742*, 2019.

[liz] Lizard. `https://github.com/terryyin/lizard/`. Accessed: 2022-11-10.

[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*, 2018.

[LLA23] Ruiyin Li, Peng Liang, and Paris Avgeriou. Code reviewer recommendation for architecture violations: An exploratory study. In *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering, EASE*, page 42–51, 2023.

[LLG+22] Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, and Neel Sundaresan. Automating code review activities by large-scale pre-training. In *30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE*, pages 1035–1047, 2022.

[LLZ+21] Jinfeng Lin, Yalin Liu, Qingkai Zeng, Meng Jiang, and Jane Cleland-Huang. Traceability transformed: Generating more accurate links with pre-trained bert models. In *43th IEEE/ACM International Conference on Software Engineering, ICSE*, pages 324–335, 2021.

[LLZJ20] Fang Liu, Ge Li, Yunfei Zhao, and Zhi Jin. Multi-task learning based pre-trained language model for code completion. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE*, pages 473–485, 2020.

[LM19] Alexander LeClair and Collin McMillan. Recommendations for datasets for source code summarization. In *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT*, pages 3931–3937, 2019.

[LNNN15] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. Combining deep learning with information retrieval to localize buggy files for bug reports (n). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE*, pages 476–481, 2015.

[LPM+21] Mingwei Liu, Xin Peng, Andrian Marcus, Christoph Treude, Xuefang Bai, Gang Lyu, Jiazhan Xie, and Xiaoxin Zhang. Learning-based extraction of first-order logic representations of api directives. In *29th ACM Joint European Software Engineering Conference and the ACM/SIGSOFT International Symposium on the Foundations of Software Engineering ESEC-FSE*, pages 491–502, 2021.

[LWW+19] Zhifang Liao, Zexuan Wu, Jinsong Wu, Yan Zhang, Junyi Liu, and Jun Long. Tirr: A code reviewer recommendation algorithm with topic model and reviewer influence. In *2019 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6. IEEE, 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 *33rd IEEE/ACM International Conference on Automated Software Engineering, ASE*, pages 373–384, 2018.

[LYJ+22] Lingwei Li, Li Yang, Huaxi Jiang, Jun Yan, Tiejian Luo, Zihan Hua, Geng Liang, and Chun Zuo. Auger: Automatically generating review comments with pre-training models. In *30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE*, page 1009–1021, 2022.

[LYY+17] Zhixing Li, Yue Yu, Gang Yin, Tao Wang, Qiang Fan, and Huaimin Wang. Automatic classification of review comments in pull-based development model. In *SEKE*, pages 572–577, 2017.

[LZS21] Xiaozhou Li, Zheying Zhang, and Kostas Stefanidis. A data-driven approach for video game playability analysis based on players' reviews. *Information*, 12(3):129, 2021.

[MAPB21] Antonio Mastropaolo, Emad Aghajani, Luca Pascarella, and Gabriele Bavota. An empirical study on code comment completion. In *37th IEEE International Conference on Software Maintenance and Evolution, ICSME*, pages 159–170, 2021.

[mar] Video games : Industry trends, monetisation strategies & market size 2020-2025 `https://www.juniperresearch.com/researchstore/content-digital-media/video-games-market-report`.

[MBN19] Chandra Maddila, Chetan Bansal, and Nachiappan Nagappan. Predicting pull request completion time: a case study on large scale cloud services. In *Proceedings of the 2019 27th acm joint meeting on european software engineering*

conference and symposium on the foundations of software engineering, pages 874–882, 2019.

[MC08] Parastoo Mohagheghi and Reidar Conradi. An empirical investigation of software reuse benefits in a large telecom product. *ACM Trans. Softw. Eng. Methodol.*, 17(3), 2008.

[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, TSE*, pages 1–20, 2022.

[MH21] 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.

[Mit97] Thomas Mitchell. *Machine Learning*. McGraw Hill, 1997.

[MKAH14] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. In *11th IEEE/ACM Working Conference on Mining Software Repositories, MSR*, pages 192–201, 2014.

[MKS+13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[MKS+15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[ML09] Mika V. Mäntylä and Casper Lassenius. What types of defects are really discovered in code reviews? *IEEE Transactions on Software Engineering*, 35(3):430–448, 2009.

[MLM+19] Vadim Markovtsev, Waren Long, Hugo Mougard, Konstantin Slavnov, and Egor Bulychev. Style-analyzer: fixing code style inconsistencies with interpretable unsupervised algorithms. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 468–478. IEEE, 2019.

[MMK15] Rodrigo Morales, Shane McIntosh, and Foutse Khomh. Do code review practices impact design quality? a case study of the qt, vtk, and itk projects. In *Proc. of the 22nd Int'l Conf. on Software Analysis, Evolution, and Reengineering (SANER)*, pages 171–180, 2015.

[MMPT21] Leonardo Mariani, Ali Mohebbi, Mauro Pezzè, and Valerio Terragni. Semantic matching of gui events for test reuse: are we there yet? In *30th ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA*, pages 177–190, 2021.

[MMY+21] Martin Monperrus, Matias Martinez, He Ye, Fernanda Madeiral, Thomas Durieux, and Zhongxing Yu. Megadiff: A Dataset of 600k Java Source Code Changes Categorized by Diff Size. Technical report, Arxiv, 2021.

[MPB22] Antonio Mastropaolo, Luca Pascarella, and Gabriele Bavota. Using deep learning to generate complete log statements. In *44th IEEE/ACM International Conference on Software Engineering, ICSE*, pages 2279–2290, 2022.

[MR20] Ehsan Mirsaeedi and Peter C. Rigby. Mitigating turnover with code review recommendation: Balancing expertise, workload, and knowledge distribution. In *42nd ACM/IEEE International Conference on Software Engineering, ICSE*, page 1183–1195, 2020.

[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*, pages 336–347, 2021.

[muj] μjava. `https://cs.gmu.edu/~offutt/mujava/`. Accessed: 2022-11-28.

[mvn22] Maven Central Repository. `https://mvnrepository.com/`, 2022.

[MYG17] Massimiliano Menarini, Yan Yan, and William G Griswold. Semantics-assisted code review: An efficient tool chain and a user study. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 554–565. IEEE, 2017.

[NDRDS+21] Phuong T. Nguyen, Juri Di Rocco, Claudio Di Sipio, Davide Di Ruscio, and Massimiliano Di Penta. Recommending api function calls and code snippets to support software development. *IEEE Transactions on Software Engineering*, 2021.

[OKI16] Ali Ouni, Raula Gaikovina Kula, and Katsuro Inoue. Search-based peer reviewers recommendation in modern code review. In *32nd IEEE International Conference on Software Maintenance and Evolution, ICSME*, pages 367–377, 2016.

[opea] Opennmt-tf. `https://github.com/OpenNMT/OpenNMT-tf/`. Accessed: 2022-11-10.

[opeb] Pit mutation operators. `https://pitest.org/quickstart/mutators/`.

[Pac] Mspacman. https://gym.openai.com/envs/MsPacman-v0/.

[PE07] Carlos Pacheco and Michael D. Ernst. Randoop: feedback-directed random testing for Java. In *ACM/SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, OOPSLA*, pages 815–816, 2007.

[PGB+21] Kexin Pei, Jonas Guan, Matthew Broughton, Zhongtian Chen, Songchen Yao, David Williams-King, Vikas Ummadisetty, Junfeng Yang, Baishakhi Ray, and Suman Jana. Stateformer: fine-grained type recovery from binaries using generative state modeling. In *29th ACM Joint European Software Engineering Conference and the ACM/SIGSOFT International Symposium on the Foundations of Software Engineering ESEC-FSE*, pages 690–702, 2021.

[pit] Pit. https://pitest.org. Accessed: 2022-11-28.

[PLV+20] Johannes Pfau, Antonios Liapis, Georg Volkmar, Georgios N Yannakakis, and Rainer Malaka. Dungeons & replicants: automated game balancing via deep player behavior modeling. In *2020 IEEE Conference on Games (CoG)*, pages 431–438, 2020.

[PLW+21] Han Peng, Ge Li, Wenhan Wang, YunFei Zhao, and Zhi Jin. Integrating tree path in transformer for code representation. In *34th Advances in Neural Information Processing Systems NIPS*, pages 9343–9354, 2021.

[PLX21] Cong Pan, Minyan Lu, and Biao Xu. An empirical study on software defect prediction using codebert model. *Applied Sciences*, 11(11):4793, 2021.

[Por80] M. F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[PPG21] Cristiano Politowski, Fabio Petrillo, and Yann-Gäel Guéhéneuc. A survey of video game testing. *arXiv preprint arXiv:2103.06431*, 2021.

[PPPB18] Luca Pascarella, Fabio Palomba, Massimiliano Di Penta, and Alberto Bacchelli. How is video game development different from software development in open source? In Andy Zaidman, Yasutaka Kamei, and Emily Hill, editors, *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, pages 392–402. ACM, 2018.

[pre] Prettier. https://prettier.io. Accessed: 2023-03-25.

[PRWZ02] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: A method for automatic evaluation of machine translation. In *40th Annual Meeting on Association for Computational Linguistics, ACL*, pages 311–318, 2002.

[PSM17] Johannes Pfau, Jan David Smeddinck, and Rainer Malaka. Automated game testing with icarus: Intelligent completion of adventure riddles via unsupervised solving. In *Extended Abstracts Publication of the Annual Symposium on Computer-Human Interaction in Play*, pages 153–164, 2017.

[PSP+18] Luca Pascarella, Davide Spadini, Fabio Palomba, Magiel Bruntink, and Alberto Bacchelli. Information needs in contemporary code review. 2(CSCW), 2018.

[PT22] Prahar Pandya and Saurabh Tiwari. Corms: A github and gerrit based hybrid code reviewer recommendation approach for modern code review. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, pages 546—-557, 2022.

[PTPI14] Thai Pangsakulyanont, Patanamon Thongtanunam, Daniel Port, and Hajimu Iida. Assessing mcr discussion usefulness using semantic similarity. In *2014 6th International Workshop on Empirical Software Engineering in Practice*, pages 49–54. IEEE, 2014.

[PTTC23] Chanathip Pornprasit, Chakkrit Tantithamthavorn, Patanamon Thongtanunam, and Chunyang Chen. D-act: Towards diff-aware code transformation for code review under a time-wise evaluation. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 296–307, 2023.

[pyc] Python bindings to the compact language detector. `https://pypi.org/project/pycld3/`. Accessed: 2022-11-28.

[PyS] Pysupertuxkart. `https://github.com/supertuxkart/stk-code`.

[QCY21] Yu Qu, Jianlei Chi, and Heng Yin. Leveraging developer information for efficient effort-aware bug prediction. *Inf. Softw. Technol.*, 137:106605, 2021.

[RAM+20] Soumaya Rebai, Abderrahmen Amich, Somayeh Molaei, Marouane Kessentini, and Rick Kazman. Multi-objective code reviewer recommendations: Balancing expertise, availability and collaborations. *Automated Software Engineering*, page 301–328, 2020.

[RB13] Peter C. Rigby and Christian Bird. Convergent contemporary software peer review practices. In *21st ACM/SIGSOFT Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering, ESEC-FSE*, pages 202–212, 2013.

[repa] `https://github.com/CodeReviewAutomationSota/code_review_automation_sota`.

[repb] `https://github.com/RosaliaTufano/api_replacement`.

[repc] `https://github.com/RosaliaTufano/code_review`.

[repd] `https://github.com/RosaliaTufano/code_review_automation`.

[repe] `https://github.com/RosaliaTufano/impact_pre-training`.

[repf] https://github.com/RosaliaTufano/impact_pre-training_code_review.

[repg] https://github.com/RosaliaTufano/rlgameauthors.

[RGCS14] Peter C. Rigby, Daniel M. Germán, Laura L. E. Cowen, and Margaret-Anne D. Storey. Peer review on open-source software projects: Parameters, statistical models, and theory. *ACM Trans. Softw. Eng. Methodol.*, 23(4):35:1–35:33, 2014.

[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 *41st IEEE/ACM International Conference on Software Engineering: New Ideas and Emerging Results, ICSE-NIER*, pages 29–32, 2019.

[RK04] Reuven Y. Rubinstein and Dirk P. Kroese. *The Cross Entropy Method: A Unified Approach To Combinatorial Optimization, Monte-Carlo Simulation (Information Science and Statistics)*. Springer-Verlag, 2004.

[RKN22] Shadikur Rahman, Umme Ayman Koana, and Maleknaz Nayebi. Example driven code review explanation. In *Proceedings of the 16th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 307–312, 2022.

[Rob95] Anthony Robins. Catastrophic forgetting, rehearsal and pseudorehearsal. *Connection Science*, 7(2):123–146, 1995.

[Ros11a] B. Rosner. *Fundamentals of Biostatistics*. Brooks/Cole, 2011.

[Ros11b] B. Rosner. *Fundamentals of Biostatistics*. Brooks/Cole, Boston, MA, 7th edition edition, 2011.

[RRC16] Mohammad Masudur Rahman, Chanchal K. Roy, and Jason A. Collins. Correct: code reviewer recommendation in github based on cross-project and technology experience. In *38th International Conference on Software Engineering, ICSE*, pages 222–231, 2016.

[RRK17] Mohammad Masudur Rahman, Chanchal K Roy, and Raula G Kula. Predicting usefulness of code review comments using textual features and developer experience. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 215–226. IEEE, 2017.

[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. *J. Mach. Learn. Res.*, 21:140:1–140:67, 2020.

[RVY14] Veselin Raychev, Martin T. Vechev, and Eran Yahav. Code completion with statistical language models. In *36th ACM/SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, pages 419–428, 2014.

[RWZ09] Martin Robillard, Robert Walker, and Thomas Zimmermann. Recommendation systems for software engineering. *IEEE software*, 27(4):80–86, 2009.

[SB21] Nishrith Saini and Ricardo Britto. Using machine intelligence to prioritise code review requests. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 11–20. IEEE, 2021.

[Sco] Scopus. `https://www.scopus.com/`. Accessed: 2022-11-15.

[SDFS20] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. Intellicode compose: code generation using transformer. In *28th ACM Joint European Software Engineering Conference and the ACM/SIGSOFT International Symposium on the Foundations of Software Engineering ESEC-FSE*, pages 1433–1443, 2020.

[SDR+18] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. Learning loop invariants for program verification. In *Neural Information Processing Systems*, 2018.

[SEB16] Behjat Soltanifar, Atakan Erdem, and Ayse Bener. Predicting defectiveness of software patches. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 1–10, 2016.

[SGBU20] Anton Strand, Markus Gunnarson, Ricardo Britto, and Muhammad Usman. Using a context-aware approach to recommend code reviewers: findings from an industrial case study. In *42nd International Conference on Software Engineering, Software Engineering in Practice, ICSE-SEIP*, pages 1–10, 2020.

[SGF+20] Jing Kai Siow, Cuiyun Gao, Lingling Fan, Sen Chen, and Yang Liu. Core: Automating review recommendation for code changes. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 284–295. IEEE, 2020.

[SLA12] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *26th Advances in Neural Information Processing Systems NIPS*, pages 2951–2959, 2012.

[SLL+19] Shu-Ting Shi, Ming Li, David Lo, Ferdian Thung, and Xuan Huo. Automatic code review by learning the revision of source code. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019*, pages 4910–4917, 2019.

[SNM09] Adam M. Smith, Mark J. Nelson, and Michael Mateas. Computational support for play testing game sketches. In *Proceedings of the Fifth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, pages 167–172, 2009.

[SNMB20] Samantha N. Stahlke, Atiya Nova, and Pejman Mirza-Babaei. Artificial players in the design process: Developing an automated testing tool for game level and world design. *Proceedings of the Annual Symposium on Computer-Human Interaction in Play*, 2020.

[sod] Stack exchange dumps. `https://archive.org/details/stackexchange`.

[Spe04] C. Spearman. The proof and measurement of association between two things. *American Journal of Psychology*, 15:88–103, 1904.

[Spr] Springer link online library. `https://link.springer.com/`. Accessed: 2022-11-15.

[srca] Scrml website. `https://www.srcml.org/`. Accessed: 2022-11-10.

[srcb] src2abs: an abstraction tool. `https://github.com/micheletufano/src2abs/`. Accessed: 2022-11-10.

[SS19] Shipra Sharma and Balwinder Sodhi. Using stack overflow content to assist in code review. *Software: Practice and Experience*, 49(8):1255–1277, 2019.

[SS23] Oussama Ben Sghaier and Houari Sahraoui. A multi-step learning approach to assist code review. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 450–460, 2023.

[SSC+18] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. Modern code review: A case study at google. In *40th International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP*, pages 181–190, 2018.

[SSH+22] Qianhua Shan, David Sukhdeo, Qianying Huang, Seth Rogers, Lawrence Chen, Elise Paradis, Peter C Rigby, and Nachiappan Nagappan. Using nudges to accelerate code reviews at scale. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 472–482, 2022.

[STD19] Emre Sülün, Eray Tüzün, and Uğur Doğrusöz. Reviewer recommendation using software artifact traceability graphs. In *Proceedings of the fifteenth international conference on predictive models and data analytics in software engineering*, pages 66–75, 2019.

[STDB23] Jaydeb Sarker, Asif Kamal Turzo, Ming Dong, and Amiangshu Bosu. Automated identification of toxic code reviews using toxicr. *ACM Transactions on Software Engineering and Methodology*, 2023.

[sup] supertuxkart. `https://github.com/supertuxkart`.

[SXP+21] Jiamou Sun, Zhenchang Xing, Xin Peng, Xiwei Xu, and Liming Zhu. Task-oriented api usage examples prompting powered by programming task knowledge graph. In *37th IEEE International Conference on Software Maintenance and Evolution ICSME*, pages 448–459, 2021.

[TdAA21] Andrew Truelove, Eduardo Santana de Almeida, and Iftekhar Ahmed. We'll fix it in post: What do bug fixes in video game update notes tell us? In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 736–747, 2021.

[TDS+20] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. Unit test case generation with transformers. *CoRR*, abs/2009.05617, 2020.

[Thu16] Ferdian Thung. Api recommendation system for software development. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 896–899, 2016.

[TK15] Yida Tao and Sunghun Kim. Partitioning composite code changes to facilitate code review. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 180–190. IEEE, 2015.

[TKD22] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. Refactoringminer 2.0. *IEEE Trans. Software Eng.*, 48(3):930–950, 2022.

[TLG+21] Ze Tang, Chuanyi Li, Jidong Ge, Xiaoyu Shen, Zheling Zhu, and Bin Luo. Ast-transformer: Encoding abstract syntax trees efficiently for code summarization. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE*, pages 1193–1195, 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 International Conference on Software Engineering, ICSE*, pages 2291–2302, 2022.

[TPB+17] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. There and back again: Can you compile that snapshot? *J. Softw. Evol. Process.*, 29(4):e1838, 2017.

[TPB23] Rosalia Tufano, Luca Pascarella, and Gabriele Bavota. Automating code-related tasks through transformers: The impact of pre-training. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, page 2425–2437, 2023.

[TPT+21] Rosalia Tufano, Luca Pascarella, Michele Tufano, Denys Poshyvanyk, and Gabriele Bavota. Towards automating code review activities. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE*, pages 163–174, 2021.

[TPT22a] Patanamon Thongtanunam, Chanathip Pornprasit, and Chakkrit Tantithamthavorn. Autotransform: Automated code transformation to support modern code review process. In *44th IEEE/ACM International Conference on Software Engineering, ICSE*, pages 237–248, 2022.

[TPT22b] 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 *41st IEEE/ACM International Conference on Software Engineering, ICSE*, pages 25–36, 2019.

[TT22] Fuwei Tian and Christoph Treude. Adding context to source code representations for deep learning. In *IEEE International Conference on Software Maintenance and Evolution, ICSME*, pages 374–378, 2022.

[TTDE21] K Ayberk Tecimer, Eray Tüzün, Hamdi Dibeklioglu, and Hakan Erdogmus. Detection and elimination of systematic labeling bias in code reviewer recommendation systems. In *Evaluation and Assessment in Software Engineering*, pages 181–190. 2021.

[TTK+15] Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Raula Gaikovina Kula, Norihiro Yoshida, Hajimu Iida, and Ken-ichi Matsumoto. Who should review my code? A file location-based code-reviewer recommendation approach for modern code review. In *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER*, pages 141–150, 2015.

[TWB+19] 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.

[UBC+21] Anderson Uchôa, Caio Barbosa, Daniel Coutinho, Willian Oizumi, Wesley KG Assunçao, Silvia Regina Vergilio, Juliana Alves Pereira, Anderson Oliveira, and

Alessandro Garcia. Predicting design impactful changes in modern code review: A large-scale empirical study. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 471–482. IEEE, 2021.

[UIIM19] Yuki Ueda, Takashi Ishio, Akinori Ihara, and Kenichi Matsumoto. Mining source code improvement patterns from similar code review works. In *2019 IEEE 13th International Workshop on Software Clones (IWSC)*, pages 13–19. IEEE, 2019.

[VEB+17] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, P. Georgiev, A. S. Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, J. Agapiou, Julian Schrittwieser, John Quan, Stephen Gaffney, S. Petersen, K. Simonyan, T. Schaul, H. V. Hasselt, D. Silver, T. Lillicrap, Kevin Calderone, Paul Keet, Anthony Brunasso, D. Lawrence, Anders Ekermo, J. Repp, and Rodney Tsing. Starcraft ii: A new challenge for reinforcement learning. *ArXiv*, abs/1708.04782, 2017.

[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. In *30th Advances in Neural Information Processing Systems NIPS*, pages 5998–6008, 2017.

[WBN21] Song Wang, Chetan Bansal, and Nachiappan Nagappan. Large-scale intent analysis for identifying large-review-effort code changes. *Information and Software Technology*, 130:106408, 2021.

[WCX+20] Yuechen Wu, Yingfeng Chen, Xiaofei Xie, Bing Yu, Changjie Fan, and Lei Ma. Regression testing of massively multiplayer online role-playing games. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 692–696. IEEE, 2020.

[wek] Weka. `http://www.cs.waikato.ac.nz/ml/weka/`. Accessed: 2022-11-10.

[WGRM18] Ruiyin Wen, David Gilbert, Michael G Roche, and Shane McIntosh. Blimp tracer: Integrating build impact analysis with code review. In *2018 IEEE International conference on software maintenance and evolution (ICSME)*, pages 685–694. IEEE, 2018.

[Whi15] Martin White. Deep representations for software engineering. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 781–783. IEEE, 2015.

[Wil] Wiley online library. `https://onlinelibrary.wiley.com/`. Accessed: 2022-11-15.

[Wil45] Frank Wilcoxon. Individual comparisons by ranking methods. *International Biometric Society, Wiley*, 1(6):80–83, 1945.

[WKIM21] Dong Wang, Raula Gaikovina Kula, Takashi Ishio, and Kenichi Matsumoto. Automatic patch linkage detection in code review using textual content and file location features. *Information and Software Technology*, 139:106637, 2021.

[WLX+19] Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. Code generation as a dual task of code summarization. In *32nd Advances in Neural Information Processing Systems NIPS, Annual Conference on Neural Information Processing Systems NeurIPS*, pages 6559–6569, 2019.

[WLZ22] Bingting Wu, Bin Liang, and Xiaofang Zhang. Turn tree into graph: Automatic code review via simplified ast driven graph convolutional network. *Knowledge-Based Systems*, 252:109450, 2022.

[WLZX19] Min Wang, Zeqi Lin, Yanzhen Zou, and Bing Xie. Cora: Decomposing and describing tangled code changes for reviewer. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1050–1061. IEEE, 2019.

[WSS17] Ke Wang, Rishabh Singh, and Zhendong Su. Dynamic neural program embedding for program repair. *arXiv preprint arXiv:1711.07163*, 2017.

[WTM+20] Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. On learning meaningful assert statements for unit test cases. In *42nd IEEE/ACM International Conference on Software Engineering, ICSE*, pages 1398–1409, 2020.

[WVVP15] Martin White, Christopher Vendome, Mario Linares Vásquez, and Denys Poshyvanyk. Toward deep learning software repositories. In *12th IEEE/ACM Working Conference on Mining Software Repositories, MSR*, pages 334–345, 2015.

[WZ22] Bingting Wu and Xiaofang Zhang. Contrastive learning for multi-modal automatic code review. *arXiv preprint arXiv:2205.14289*, 2022.

[XLWY15] Xin Xia, David Lo, Xinyu Wang, and Xiaohu Yang. Who should review this change?: Putting text and file location analyses together for more accurate recommendations. In *31th IEEE International Conference on Software Maintenance and Evolution, ICSME*, pages 261–270, 2015.

[XMYI14] Pei Xia, Makoto Matsushita, Norihiro Yoshida, and Katsuro Inoue. Studying reuse of out-dated third-party code in open source projects. *Information and Media Technologies*, 9(2):155–161, 2014.

[XSJ+17] Zhenglin Xia, Hailong Sun, Jing Jiang, Xu Wang, and Xudong Liu. A hybrid approach to code reviewer recommendation with collaborative filtering. In *6th International Workshop on Software Mining, SoftwareMining*, pages 24–31, 2017.

[YCLW16] Haochao Ying, Liang Chen, Tingting Liang, and Jian Wu. Earec: leveraging expertise and authority for pull-request reviewer recommendation in github. In *3rd International Workshop on CrowdSourcing in Software Engineering, CSI-SE@ICSE*, pages 29–35, 2016.

[Ye19] Xin Ye. Learning to rank reviewers for pull requests. *IEEE Access*, pages 85382–85391, 2019.

[YKY+21] Zhen Yang, Jacky Keung, Xiao Yu, Xiaodong Gu, Zhengyuan Wei, Xiaoxue Ma, and Miao Zhang. A multi-modal transformer-based code summarization approach for smart contracts. In *29th IEEE/ACM International Conference on Program Comprehension, ICPC*, pages 1–12, 2021.

[YWW22] Shiwen Yu, Ting Wang, and Ji Wang. Data augmentation by program transformation. *J. Syst. Softw.*, 190:111304, 2022.

[YWYW16] Yue Yu, Huaimin Wang, Gang Yin, and Tao Wang. Reviewer recommendation for pull-requests in github: What can we learn from code review and bug assignment? *Inf. Softw. Technol.*, 74:204–218, 2016.

[ZBB+19] Yunqi Zhao, Igor Borovikov, Ahmad Beirami, Jason Rupert, Caedmon Somers, Jesse Harder, Fernando de Mesentier Silva, John Kolen, Jervis Pinto, Reza Pourabolghasem, Harold Chaput, James Pestrak, Mohsen Sardari, Long Lin, Navid Aghdaie, and Kazi A. Zaman. Winning isn't everything: Training human-like agents for playtesting and game ai. *CoRR*, abs/1903.10545, 2019.

[ZdCZ19] Guoliang Zhao, Daniel Alencar da Costa, and Ying Zou. Improving the pull requests review process using learning-to-rank algorithms. *Empirical Software Engineering*, 24:2140–2170, 2019.

[ZFR14] Alexander Zook, Eric Fruchter, and Mark O. Riedl. Automatic playtesting for game parameter tuning via active learning. *ArXiv*, abs/1908.01417, 2014.

[ZHL21] Xin Zhou, DongGyun Han, and David Lo. Assessing generalizability of codebert. In *37th IEEE International Conference on Software Maintenance and Evolution, ICSME*, pages 425–436, 2021.

[ZKB16] Motahareh Bahrami Zanjani, Huzefa Kagdi, and Christian Bird. Automatically recommending peer reviewers in modern code review. *IEEE Transactions on Software Engineering*, 42(6):530–543, 2016.

[ZLL+21] Teng Zhou, Kui Liu, Li Li, Zhe Liu, Jacques Klein, and Tegawendé F Bissyandé. Smartgift: Learning to generate practical inputs for testing smart contracts. In *37th IEEE International Conference on Software Maintenance and Evolution ICSME*, pages 23–34, 2021.

[ZMA+22] Fiorella Zampetti, Saghan Mudbhari, Venera Arnaoudova, Massimiliano Di Penta, Sebastiano Panichella, and Giuliano Antoniol. Using code reviews to automatically configure static analysis tools. *Empirical Software Engineering*, 27(1):28, 2022.

[ZMB+23] Jiyang Zhang, Chandra Maddila, Ram Bairi, Christian Bird, Ujjwal Raizada, Apoorva Agrawal, Yamini Jhawar, Kim Herzig, and Arie van Deursen. Using large-scale heterogeneous graph representation learning for code review recommendations at microsoft. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 162–172, 2023.

[ZMW19] Sarim Zafar, Muhammad Zubair Malik, and Gursimran Singh Walia. Towards standardizing and improving classification of bug-fix commits. In *13th International Symposium on Empirical Software Engineering and Measurement, ESEM*, pages 1–6, 2019.

[ZPW+21] Jiayuan Zhou, Michael Pacheco, Zhiyuan Wan, Xin Xia, David Lo, Yuan Wang, and Ahmed E Hassan. Finding a needle in a haystack: Automated mining of silent vulnerability fixes. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE*, pages 705–716, 2021.

[ZXS+19] Y. Zheng, X. Xie, T. Su, L. Ma, J. Hao, Z. Meng, Y. Liu, R. Shen, Y. Chen, and C. Fan. Wuji: Automatic online combat game testing using evolutionary deep reinforcement learning. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 772–784, 2019.

[ZYK+22] Fengji Zhang, Xiao Yu, Jacky Keung, Fuyang Li, Zhiwen Xie, Zhen Yang, Caoyuan Ma, and Zhimin Zhang. Improving stack overflow question title generation with copying enhanced codebert model and bi-modal information. *Information and Software Technology*, 148:106922, 2022.

[ZZSL20] 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.