

Towards Automatically Addressing Self-Admitted Technical Debt: How Far Are We?

Antonio Mastropaolo
SEART @ Software Institute,
Università della Svizzera italiana (USI),
Switzerland

Massimiliano Di Penta
Dept. of Engineering,
University of Sannio,
Italy

Gabriele Bavota
SEART @ Software Institute,
Università della Svizzera italiana (USI),
Switzerland

Abstract—Upon evolving their software, organizations and individual developers have to spend a substantial effort to pay back technical debt, *i.e.*, the fact that software is released in a shape not as good as it should be, *e.g.*, in terms of functionality, reliability, or maintainability. This paper empirically investigates the extent to which technical debt can be automatically paid back by neural-based generative models, and in particular models exploiting different strategies for pre-training and fine-tuning. We start by extracting a dataset of 5,039 Self-Admitted Technical Debt (SATD) removals from 595 open-source projects. SATD refers to technical debt instances documented (*e.g.*, via code comments) by developers. We use this dataset to experiment with seven different generative deep learning (DL) model configurations. Specifically, we compare transformers pre-trained and fine-tuned with different combinations of training objectives, including the fixing of generic code changes, SATD removals, and SATD-comment prompt tuning. Also, we investigate the applicability in this context of a recently-available Large Language Model (LLM)-based chat bot. Results of our study indicate that the automated repayment of SATD is a challenging task, with the best model we experimented with able to automatically fix $\sim 2\%$ to 8% of test instances, depending on the number of attempts it is allowed to make. Given the limited size of the fine-tuning dataset ($\sim 5k$ instances), the model’s pre-training plays a fundamental role in boosting performance. Also, the ability to remove SATD steadily drops if the comment documenting the SATD is not provided as input to the model. Finally, we found general-purpose LLMs to not be a competitive approach for addressing SATD.

Index Terms—Self-Admitted Technical Debt, Pre-trained models, and Machine Learning for Code

I. INTRODUCTION

Technical Debt (TD) has been defined by Cunningham as “not-quite-right code” [1]. Essentially, the terms refer to the debt an organization or an individual developing and releasing software should repay to make it acceptable, for example in terms of functionality, reliability, or maintainability. Oftentimes, developers achieve awareness of the TD in their program, admitting it through comments, commit messages, or issues. This has been referred to as “Self-Admitted Technical Debt” (SATD) [2]. Previous research has investigated why developers annotate software as SATD, essentially to keep track of what needs to be improved [3]. Also, the analysis of SATD in existing programs has shown how developers take it seriously, as it gets removed in the majority of the cases [4], even though this often happens when the source code is completely replaced or even removed [5]. As previous work

found [6], [7], [2], [3], SATD relates to different problems in the program, such as the need to fix bugs occurring in certain circumstances, enhancing or even completing a feature, or improving the source code maintainability and quality in general.

Researchers have proposed various kinds of approaches to aid developers with the management of SATD. On the one hand, while SATD comments are usually recognizable by commonly used keywords such as “TODO” or “FIXME”, this is not always the case. Therefore, approaches leveraging several types of techniques ranging from simple regular expression matching [2] to shallow machine learning [8], and deep learning [9] have been used to identify SATD comments.

Also, some approaches recommend developers with the type of change that needs to be carried out to address the SATD [10]. While previous work has proposed approaches to guide developers towards paying back TD, to the best of our knowledge there is no specific work aimed at automatically resolving (SA)TD. Indeed, this can be a direction worthwhile to investigate, considering the significant advances in the application of generative approaches to software-related tasks, such as code completion [11], [12], [13], [14], [15], program repair [16], [17], [18], [19], vulnerability patching [20], [21], or code review [22], [23], [24], [25]. However, given the diversity of SATD, its repayment would require approaches that go beyond what has been already devised for each of the aforementioned tasks. Therefore, this paper aims to answer the following question:

Can AI-based approaches automatically repay the technical debt?

To answer this question, we investigate the extent to which neural generative approaches based on deep learning transformers [26] can be used to repay SATD. An obvious question that can arise is whether SATD resolution is, in the end, equivalent to program repair. We believe there are a series of differences and challenges:

- 1) As also pointed out by previous work [3], repaying (SA)TD not only means fixing bugs, but it also requires to improve the code in different ways, for example enhancing a feature, making the code able to handle certain special scenarios, or adopting alternative APIs.
- 2) Differently from other “buggy” code, SATD admits the presence of a problem, therefore directly highlighting the

portions of the source code that need to be repaired.

- 3) Neural models require a conspicuous amount of training data, which may be available for certain tasks (*e.g.*, code completion), and less for others, including SATD resolution.

To study how SATD can be addressed by generative models, we first created a dataset of 5,039 SATD removal instances from 595 open-source projects by leveraging an available tool that detects SATD removals [27]. Then, we leverage a pre-trained transformer model (CodeT5 [28]) that previous work showed particularly effective to cope with problems where the size of the training set is limited [21], [29]. Through different experiments, we test the effectiveness of several pre-training/fine-tuning strategies, including:

- 1) No pre-training, fine-tuning using SATD removal instances.
- 2) Self-supervised pre-training followed by fine-tuning on SATD removals. Self-supervised pre-training exploits training objectives not requiring a supervised dataset. We use the *masked language model* objective [30], [31], which consists in providing the model with input sentences (*e.g.*, an English sentence, a *Java* method, depending on the language of interest) having 15% of their tokens masked, asking the model to predict them.
- 3) Self-supervised and supervised pre-training followed by fine-tuning on SATD removal instances. Supervised pre-training can be used to pre-train the model on a task similar to the downstream one. In our case, we pre-train the model for the implementation of generic code changes, before fine-tuning it with SATD removals.

We also experimented with the impact on the model's performance with and without the SATD comment. This is worthwhile to study because (i) the SATD comment may act as a sort of prompt-tuning for the transformer [26] which has been shown to help models pre-trained on English text (such as CodeT5); and (ii) a boost in performance motivates the usefulness of SATD comments not only as a trace for developers [3], but also as a way to aid AI-based approaches. Finally, we experiment with the extent to which SATD can be addressed by leveraging a Large Language Model (LLM) chat bot, *i.e.*, ChatGPT [32].

Results of our study indicate that automatically addressing SATD instances is a challenging task, and the best model we experimented with is able to correctly address 2.30% (one attempt) to 8.10% (ten attempts) of the SATD instances in our test set. Without any pre-training, the model is not able to address any SATD instance, likely due to the limited size of the fine-tuning dataset. Self-supervised pre-training helps in improving performance, which is further increased when the model is also subject to supervised pre-trained on a task (*i.e.*, implementing generic code changes) resembling the downstream one (*i.e.*, addressing SATD). Finally, we experimented with three different prompts for ChatGPT, with the best one being able to address only 1.19% of the SATD in our dataset, confirming how challenging the tackled task is. In summary, while the studied approaches can in some cases

automatically repay SATD, there is still a long way to go to fully address this problem.

Overall, the paper contributes to the state-of-the-art on (SA)TD management and resolution with:

- 1) An experimentation featuring seven different combinations of treatments on the use of pre-trained neural transformers for SATD repayment;
- 2) Results of a study on the feasibility of using a LLM chat bot (ChatGPT) for SATD repayment; and
- 3) A replication dataset that can be also used for further experiments in this area [33].

II. RELATED WORK

Given the emphasis of our investigation, we will focus our discussion on two primary research areas: (i) methods that assist in managing and removing SATD comments, and (ii) neural models that automate program repair and code review tasks. For a comprehensive overview of DL-based models applied in software engineering, we point the reader to the systematic literature review by Watson *et al.* [34].

A. SATD Management and its Removal

Previous works have studied SATD along various dimensions, and in particular, the types of SATD occurring in programs and its relation with software quality [6], the polarity of developers' comments with relationship to different SATD types [35], or the extent to which source code not containing an SATD may indeed require it [36]. We only discuss work specifically related to SATD removal. Further research about SATD can be found in a survey by Sierra *et al.* [37].

SATD removal represents an important activity for software developers. Maldonado *et al.* [4] analyzed the SATD removal in five Java open-source projects. Their findings indicate that the majority of SATD is being removed, about half of which by the same person introducing it. A follow-up study by Zampetti *et al.* [5] performed a fine-grained analysis of SATD removals, finding that a large percentage of SATD removals occur "accidentally" along with other changes. Most of the changes required to remove SATD are complex ones, *i.e.*, requiring the addition, removal, or replacement of multiple source code lines. In some circumstances, SATD is removed by means of specific changes, *e.g.*, to conditionals in control-flow statements or to APIs. In general, results of Zampetti *et al.* indicate that, beyond the cases where the code simply disappears, the type and span of changes required to address a SATD vary a lot, requiring in some cases simple code edits (*e.g.*, change a condition), and in other cases the addition of new blocks of code (*e.g.*, add new behavior), with the changes sometimes spanning in multiple places. This, unavoidably, makes the automated SATD fix a very challenging task.

Despite previous work suggested that SATD is often removed "by chance", Tan *et al.* [38] surveyed developers to understand whether they intentionally fix SATD. Their results indicate that in the majority of the cases, developers are conscious of the SATD removal and balance the pros and cons of this action. On the same line, Pina *et al.* [39] studied,

through a survey, how developers prioritize technical debt management finding that, when developers decide to replay technical debt, they do it at their earliest convenience. Overall, the aforementioned work motivates ours.

The first work aimed at suggesting how to address SATD was proposed by Zampetti *et al.* [10]. They leveraged a model taking as input the SATD comment and the affected source code, to determine the type of change to apply—along six categories—for repaying the SATD. Differently from Zampetti *et al.* [10], we propose to automatically recommend the actual fix instead of suggesting its category. This is made possible by the use of pre-trained models.

B. Neural Models for Automated Program Repair and Review

Researchers have leveraged various types of neural models for Automated Program Repair (APR). Chen *et al.* [40] proposed SequenceR, which is a vanilla version of Transformer with copy mechanism to handle the out-of-vocabulary problem (OOV), correctly predicting 950/4,711 (~20%) fixes for the buggy component provided as input to the model. Tufano *et al.* [41] investigated the usage of Neural Machine Translation (NMT) to generate patches for buggy code, specifically *Java* methods. Their approach was trained using buggy methods provided as input, which were “translated” into their respective fixed versions. In a separate work, Tufano *et al.* [42] leveraged a different NMT model to automatically apply code changes implemented by developers during pull requests (PRs).

Jiang *et al.* [17] suggested CURE, a GPT (Generative-Pretrained-Transformer) architecture [43] pre-trained on source code and using a subword tokenization technique to generate a smaller search space containing more correct fixes.

Mashhadi and Hemmati [16] proposed the usage of CodeBERT [44] to fix simple bugs in the context of *Java* programs. Once the model has been fine-tuned on the ManySSuBs4J dataset [45], Mashhadi and Hemmati assessed its capabilities in generating meaningful fixes, finding that the devised approach can produce fixes for different types of bugs as a real developer would do in up to 72% of cases.

Mastropaolo *et al.* [46], [47] explored the capability of the Text-to-Text Transfer Transformer (T5) model in supporting various code-related tasks, including bug-fixing.

Lutellier *et al.* [18] introduced CoCoNuT, a CNN-driven NMT model employing ensemble learning for producing bug fixes. The assessment conducted on four languages (Java, C, Python, and JavaScript) revealed that CoCoNuT successfully resolved 509 bugs, including 309 bugs not previously addressed by existing methods.

Li *et al.* [19] proposed a tree-based RNN architecture to generate bug fixes. The usage of tree-based RNN allows modeling code and at the same time to learn tree-based structural code information from past bug fixes.

Concerning the automation of code review activities on code, Tufano *et al.* [22] made a first step towards automating code review tasks by utilizing Transformer models to implement code changes as requested by reviewers. In a subsequent study, Tufano *et al.* [23] further explored the use of pre-trained

models and BPE-like schema tokenization [48], to advance the state-of-the-art of code review automation. Thongtanunam *et al.* [49] also confirmed the advantages of Transformer-based models and BPE-tokenization for improving the performance of DL-based systems in automating code reviews. Other research in this domain includes CodeReviewer [24], which utilizes a Transformer encoder-decoder model pre-trained with four tailored tasks for code review.

Given existing APR and code review research we empirically investigate how state-of-the-art approaches can be adapted to repay (SA)TD.

Different authors have proposed transformer models specifically adapted to support code edit tasks. Some of these works proposed AST-specific representations able to learn and predict code edits [50], [51], [52]. However, these require the models to be trained from scratch, and to rely on an AST-based representation. A more general approach has been proposed by Zheng *et al.* [53], who introduced CodiT5. CodiT5 is based on the same architecture as CodeT5, and, given a sequence of tokens to repair, it predicts an edit plan, in terms of insert, delete, and replace operations. CodiT5 has been pre-trained on both natural language and source code and, with proper fine-tuning, showed superiority to other models (including CodeT5) for tasks such as comment updating, bug fixing, and automated code review.

Recently, LLMs such as GPT-3 [54] or GPT-4 [55] have propelled the automation of code-related tasks to new heights. Prenner and Robbes [56] investigated the extent to which the OpenAI’s Codex Model [57] is able to localize and fix bugs. The study’s results highlighted the impressive performance of LLMs in zero-shot settings, exhibiting competitive results compared to the latest state-of-the-art techniques.

We take inspiration from the latest research and explore the capability of the new ChatGPT assistant [55] introduced by OpenAI. Specifically, we conducted experiments within a closed-setting scenario (*i.e.*, zero-shot learning) to evaluate the model’s ability to generate code changes required when addressing *Self-Admitted Technical Debts*.

III. STUDY DEFINITION, DESIGN AND PLANNING

The *goal* of this study is to evaluate DL-based solutions in automatically implementing code changes required to address SATD in *Java* code. The *context* of the study features two deep learning models, namely CodeT5 [28] and ChatGPT [32], and two datasets used for pre-training and fine-tuning the experimented models. The pre-training dataset features generic code changes implemented by developers in open-source projects and has been presented in the work by Tufano *et al.* [42]. The fine-tuning dataset is a contribution of this paper and features SATD removal changes.

In the following, we formulate the study research questions (RQs) (Section III-A). Then, Section III-B describes the datasets used to train and test the experimented techniques. The latter are presented in Section III-C. We conclude by outlining the data analysis procedure in Section III-D.

A. Research Questions

Given our overall goal (*i.e.*, assessing the capabilities of DL-based solutions in automatically addressing SATD), we formulate the following RQs:

RQ₁: *To what extent do pre-trained models of code support automated SATD repayment?* In RQ1 we fine-tune the pre-trained CodeT5 [28] model for the task of SATD repayment and assess its performance. We also investigate the role played by the self-supervised pre-training on CodeT5, *i.e.*, the extent to which the self-supervised pre-training helps in fixing SATD. RQ1 provides a starting point for our investigation, showing what performance can be achieved by just fine-tuning an existing pre-trained model for SATD repayment.

RQ₂: *To what extent does the infusion of “similar-task knowledge” in pre-trained models of code benefits the automated SATD repayment?* While CodeT5 [28] has been pre-trained using the *masked language model* self-supervised objective, other forms of pre-training are possible. RQ₂ evaluates the effectiveness of performing a further pre-training step aimed at instilling in the model knowledge about a task resembling the downstream one (*i.e.*, the SATD repayment). This means that, before fine-tuning the model for SATD repayment, we leverage a supervised pre-training in which the model learns how to implement generic code changes. The rationale is that this task, while different from SATD repayment, can start driving the model’s weights toward a configuration closer to the one needed for the downstream task.

RQ₃: *To what extent does the presence of “context-specific knowledge” help pre-trained models of code in the automated SATD repayment?*

SATD instances can be represented as pairs $\langle \text{comment}, \text{code} \rangle$ where the *comment* describes the SATD to address in the *code*. When assessing the performance of DL-based solutions in automatically addressing SATD, the $\langle \text{comment}, \text{code} \rangle$ pair represents the input of the model which is expected to produce a *revised_code* addressing the technical debt. If we factor out the *comment* from the input the task becomes similar to those previously tackled in the literature through DL models, such as learning generic code changes implemented by developers [42] or fixing bugs [41], [17], [16], [18], [20]. For these tasks the model’s input is just a *code* in which a change (*e.g.*, a bug fix) must be implemented, thus producing as output the *revised_code*. RQ₃ assesses the extent to which providing the SATD *comment* to the model helps to address the TD, thus investigating whether training a SATD-specialized model is worthwhile as compared to just using a model trained to address generic code changes without relying on the SATD comment (see *e.g.*, [42]).

RQ₄: *Are general-purpose large language models zero-shot learners for SATD repayment?* In RQ4 we study whether LLMs (and, in the specific case, ChatGPT [32]) can be considered as out-of-the-box solutions for the automated SATD repayment. While, for what concerns source code ChatGPT has been “seen” the entire GitHub, it has not been fine-tuned for

the specific problem of SATD repayment. A positive answer to RQ4 would indicate that research on specialized models for SATD repayment is unlikely to be relevant/beneficial.

B. Context: Datasets

We describe the pre-training and fine-tuning datasets used in our research, which are summarized in Table I.

1) *Self-supervised pre-training on bi-modal data:* In the first three RQs, we employ CodeT5 [28] as representative of a state-of-the-art pre-trained model of code. CodeT5 is a Text-To-Text Transfer Transformer (T5) model [58] pre-trained on code and natural language (*i.e.*, code comments). Among different transformer models we have chosen CodeT5, as it has been used successfully for several tasks, including code summarization [28], source code generation [59], vulnerability patching [21], code review automation [25], code-to-code translation [60], and shown to outperform other models such as CodeBERT [44], PLBART [61] and GraphCodeBERT [62].

Wang *et al.* [28] utilized the CodeSearchNet dataset [63] for pre-training CodeT5 using the masked language model objective (*i.e.*, self-supervised pre-training by randomly masking 15% of the input asking the model to predict it). This dataset includes functions written in six different programming languages (Go, Java, JavaScript, PHP, Python, and Ruby). A subset of these functions also includes a top-level comment (*e.g.*, Javadoc for Java). In addition, Wang *et al.* gathered extra data from C/C# repositories hosted on GitHub. This led to a total of 8,347,634 pre-training code functions: 3,158,313 of these functions are paired with their documentation, while 5,189,321 consist solely of code.

2) *Supervised pre-training on generic code changes:* Tufano *et al.* [42] proposed the usage of NMT to learn how to automatically apply code changes implemented by developers during pull requests (PRs). The NMT model has been trained on a dataset featuring PRs from three *Gerrit* [64] repositories: (i) *oVirt*, (ii) *Android*, and (iii) *Google*.

Each of these repositories groups multiple projects (*e.g.*, all those related to the *Android* operating system), with the authors focusing on the ones written in *Java*. For each PR, Tufano *et al.* extracted two versions of the files involved in the change diff: The version before the PR was implemented and the version after the PR has been merged. These files have then been parsed to extract a total of 631,307 pairs of methods $\langle m_b, m_a \rangle$ representing the same method before (m_b) and after (m_a) the changes implemented in the PR. The idea was to train the NMT model on these pairs to see if it was able to learn generic code changes that developers might implement in the context of PRs.

We leverage this dataset in the context of RQ₂ to investigate whether the infusion of “similar-task knowledge” in pre-trained models of code benefits the automated SATD repayment. Starting from the $\sim 630k$ pairs in the original dataset we discarded instances containing non-ASCII tokens, and those having $\#tokens > 1024$. The latter filter is necessary to manage the computational complexity of training large DL-models and is a common practice in the software engineering

literature [65], [12], [23], [66], [22], [42]. For example, in the original work by Tufano *et al.* [42] in which this dataset has been created, the authors removed all pairs having $\#tokens > 100$. Subsequently, we eliminate duplicated pairs $\langle m_b, m_a \rangle$, obtaining a final dataset of 284,190 instances. We split the processed set of methods into 90% training and 10% validation. The former will be used to perform supervised pre-training on the task of learning generic code changes. The latter is instead employed to identify the best-performing checkpoint while performing the supervised pre-training.

TABLE I: Number of instances included in the datasets we used to train, test and evaluate the models

Dataset	Train	Test	Eval	Overall
Generic code changes [42]	255,771	-	28,419	284,190
SATD removal	3,537	1,000	502	5,039
Total	259,308	1,000	28,921	289,229

3) *Fine-tuning dataset of SATD removals*: As a first step to build the fine-tuning dataset, we collected a list of *Java* repositories leveraging the GitHub Search tool by Dabic *et al.* [67]. The querying user interface allows to identify GitHub projects that meet specific selection criteria. We selected all *Java* projects having at least 500 commits, 10 contributors, 10 stars, and not being forks (to reduce the chance of mining duplicated code). The commits/contributors/stars filters aim at discarding personal/toy projects. Instead, the decision of narrowing down the scope to only *Java* as a programming language was dictated by (i) the will of reusing the previously described *Java* dataset by Tufano *et al.* (Section III-B2) for supervised pre-training, and (ii) the usage in our toolchain of tools only supporting *Java* (e.g., *SATDBailiff*, as described in the following). We collected 6,971 *Java* repositories.

To extract changes aimed at addressing SATD instances we rely on the *SATDBailiff* tool by AlOmar *et al.* [27]. *SATDBailiff* can identify commits featuring the addition, removal, or change of SATD instances in the history of *Java* git repositories. We are interested in mining from the history of the subject *Java* repositories commits featuring removal events, since those are the ones implementing changes aimed at addressing SATD and, thus, are the ones suitable for our fine-tuning. Unfortunately, we found this extraction process to be extremely expensive. For this reason, we set two boundaries for our data mining. First, we allowed *SATDBailiff* to process each repository for at most 60 minutes. Indeed, we observed that for some very large projects the mining procedure could go on for days. If a repository could not be analyzed within 60 minutes, the repository was discarded from our study. Second, we set 50 days as the maximum boundary for the mining procedure. In this period, *SATDBailiff* successfully analyzed the entire history of 809 *Java* projects hosted on GitHub.

These 809 *Java* projects yielded a total of 519,440 SATD-related events including *SATD_REMOVED*, *SATD_ADDED*, and *SATD_CHANGED*. We extracted the 139,803 concerning *SATD_REMOVED*. Then, we further refined this list by

only selecting instances for which the SATD comment extracted by *SATDBailiff* contained specific keywords describing the presence of *Self-Admitted Technical Debts*: `to(-)do`, `fix(-)me`, `check(-)me`, `hack(-)me`, and `xxx`. In principle, we are aware that such further filtering may reduce the dataset construction recall and, ultimately, the dataset size. However, as the SATD detection performed by *SATDBailiff* is based on a ML-based approach, it is inherently subject to false positives, and we wanted to avoid experimenting with changes that were not related to SATD removal. We have chosen the aforementioned keywords since those are well-known patterns used to signal SATD [2], [35]. After this further pruning, we obtain a dataset where each instance is a triplet `commit_before`, `commit_after`, and `comment`, where the two commits indicate the version of the code affected (`commit_before`) by and cleaned from (`commit_after`) the SATD, while `comment` is the code comment documenting the SATD which is linked to a specific *Java* file.

We removed duplicated instances, namely those being characterized by the same triplet (e.g., due to the same SATD fixed in the same commit in different files). This left us with 75,083 instances which could feature the SATD in any part of the impacted *Java* file. However, we are interested in training the DL-model to address SATDs affecting a specific method, ignoring SATD instances related to e.g., class instance variables, `import` statements etc. The reason for such a choice is two-fold. First, also the pre-training datasets are defined at function-level granularity, thus suggesting a similar fine-tuning to take full advantage of the knowledge acquired during pre-training. Second, providing an entire *Java* file as input to the model makes the training extremely expensive, since the length of the input sequences will grow to tens of thousands of tokens. Thus, we parsed the code file in the `commit_before` to see if the SATD comment was within a method m_{satd} or immediately above it. This was the case for 65,380 instances.

Scenario ① depicted in Fig. 1 shows a SATD comment preceding the implementation of `ensureIndex` method, while in ② the comment documenting the TD is included within the body of the `configureOptions` method. Based on what we have explained so far, we work on the following assumption:

If an SATD comment is removed, the changes performed within the same commit and in the method to which the SATD comment was attached are related to repaying such an SATD.

Given the available dataset, our aim is to create the final training triplets $\langle m_{satd}, m_{fixed}, comment \rangle$, where $\langle m_{satd}, comment \rangle$ represents the model's input and m_{fixed} the model's output. To achieve this goal, some additional checks are required. First, as previous work has found [5], it is possible that addressing the SATD requires the deletion of m_{satd} , or the implementation of other methods while leaving m_{satd} unchanged (except for the removal of the SATD comment). Thus, we verify that (i) a method having the exact

```

Msatd types featured within our SATD Removal Dataset

@Override
//TODO: need to support compound indexes
public void ensureIndex(final String key, final OrderBy orderBy)
{
    ensureIndex(key, orderBy, false);
}

private void configureOptions() {
    LOG.debug("Auto detecting current execution environment");
    //FIXME Make it pluggable
    this.options.putExtraAttribute(
        FileIoProcessor.OPTION_EXPORTER_ENABLED,
        GenericOptionValue.AUTO.getSymbol());
}

```

Fig. 1: SATD removal instances in our fine-tuning dataset

same name of m_{satd} exists in `commit_after`, and (ii) by removing the SATD comment from m_{satd} we obtain a method $m'_{satd} \neq m_{fixed}$. The first filter guarantees that m_{satd} still exists in `commit_after`, while the second ensures that changes have been implemented in m_{satd} to address the SATD (obtaining m_{fixed}). This cleaning left us with 12,267 triplets.

We manually inspected 100 triplets to look for additional problematic cases. We found triplets characterized by “meaningless” SATD comments, such as “TODO” not followed by anything else. These comments do not really describe a SATD and, for this reason, we decided to exclude from our dataset all triplets having a *comment* featuring less than three words that are unlikely to describe a SATD in enough detail to be understood (8,564 instances left).

Finally, we used the code-tokenize Python library [68] to extract a tokenized version of the extracted methods and removed triplets featuring methods having $\#tokens > 1024$, and instances that raised errors while being tokenized. After filtering, we ended up with a total of 5,039 triplets derived from 595 Java projects, which constitute our fine-tuning dataset. The latter is further split into 70%, 20%, and 10% for training, testing, and validation of the models, respectively. These triplets have been processed to introduce two special tokens $\langle SATD_START \rangle$ and $\langle SATD_END \rangle$ which serve to tag the start and end of the SATD comment within m_{satd} . As previous work did [21], [23], the idea is that these tokens could help direct the model’s attention toward relevant sections of the input. Fig. 2 depicts an example of instance from the dataset we built.

Note that we create two different versions of the SATD removal dataset, both containing the same number of instances across training, testing, and validation.

However, while the first one also contains the SATD comment, the latter is removed in the second one. This is necessary to address RQ_3 , i.e., to determine the extent to which admitting TD would not only serve as a trace for the developers but also as an aid for automated tools.

C. Experimented Techniques

As we are interested to study the performance of different DL-based solutions for automatically addressing SATD instances, we focus on two recently presented models, namely CodeT5 [28] and ChatGPT [32]. For the former, we use the $CodeT5_{base}$ variant, featuring 220 million trainable parameters. We use the default architecture and hyperparameters

```

METHOD INTRODUCING THE SATD COMMENT (M_satd)

public void beforeTest(TestContext context) {
    if (beforeTest != null) {
        for (SequenceBeforeTest sequenceBeforeTest : beforeTest) {
            try {
                if (sequenceBeforeTest.shouldExecute(getName(),
                    getPackageName(), null))
                <SATD_START> //TODO provide test group information <SATD_END>
                    sequenceBeforeTest.execute(context);
            }
            catch (Exception e) {
                throw new CitrusRuntimeException("Before test...", e);
            }
        }
    }
}

METHOD RESOLVING THE SATD COMMENT (M_fixed)

public void beforeTest(TestContext context) {
    if (beforeTest != null) {
        for (SequenceBeforeTest sequenceBeforeTest : beforeTest) {
            try {
                if (sequenceBeforeTest.shouldExecute(getName(),
                    getPackageName(), groups))
                    sequenceBeforeTest.execute(context);
            }
            catch (Exception e) {
                throw new CitrusRuntimeException("Before test...", e);
            }
        }
    }
}

```

Fig. 2: SATD removal instance in our fine-tuning dataset

of $CodeT5_{base}$ featuring 12 Transformer Encoder blocks, 12 Transformer Decoder blocks, 768 hidden sizes, and 12 attention heads. The learning rate is set to $2e-5$.

While CodeT5 has been specifically pre-trained and fine-tuned to support software engineering tasks, ChatGPT is a general-purpose LLM designed and developed by OpenAI to produce human-like responses for a broad spectrum of language-related tasks (e.g., question-answering, language translation, coding tasks, etc.). There are currently two versions of ChatGPT, one built on top of GPT-3.5 and one exploiting GPT-4.0. Given the current restrictions on the usage of GPT-4.0, we carried out our research using GPT-3.5 as the foundational model for ChatGPT. Although considered “less proficient” than the chat-bot built using GPT-4.0, the version employed for our experiments, with 154 billion parameters model (GPT-3.5), is still in the LLM category, and it is definitely by far a larger model than CodeT5.

In the following, we detail how we used the two models to answer our RQs. In particular, we pre-trained, fine-tuned, and queried the models using several different strategies. All fine-tunings have been executed for a maximum of 50 epochs on the “SATD removal” dataset (see Table I). To cope with overfitting, we stop the fine-tuning using an early stopping procedure assessing the loss of the model on the validation set every epoch, using a delta of 0.01 and patience of 5. This means that the training process stops if a gain lower than delta (0.01) is observed after 5 consecutive epochs and the best-performing checkpoint up to that training step is selected.

1) *No Pre-training + Fine-tuning (RQ_1)*: We fine-tune on the context-specific SATD removal dataset (i.e., the one including the comment documenting the SATD) a $T5_{base}$ model [58] (i.e., the same used for CodeT5 [28]) without any pre-training. To this aim, we start by randomly initializing the weights of the model, which will be adjusted during the fine-tuning procedure. Such a model serves as a baseline to

assess the impact of different pre-trainings on the model’s performance.

2) *Self-supervised Pre-training + Fine-tuning (RQ_1):* We start from the CodeT5 model pre-trained using a self-supervised objective (i.e., *masked language model*) and fine-tune it on the context-specific SATD removal dataset.

3) *Self-supervised & Supervised Pre-training + Fine-tuning (RQ_2 and RQ_3):* Previous works in the natural language processing [69] and in the software engineering literature [20], [70] suggest that exploiting a supervised pre-training objective that resembles the downstream task (in our case, SATD repayment) can play a positive role on the models’ performance. For this reason, we further pre-trained CodeT5 for five epochs using the “generic code changes” dataset described in Section III-B2. Following that, we continue fine-tuning CodeT5, which has been enhanced with domain-specific knowledge, on the SATD removal dataset, both with and without additional context, i.e., code comments.

4) *Zero-Shot Prompt Tuning (RQ_4):* We designed three prompt templates aimed at querying ChatGPT for the SATD repaying task. All prompts feature the SATD *comment* and the method affected by SATD (m_{satd}):

- 1) Remove this SATD: {*comment*} from the following code { m_{satd} }
- 2) Perform removal of this SATD: {*comment*} from this code { m_{satd} }
- 3) This code { m_{satd} } contains the following SATD: {*comment*} remove it

We also tried to explain ChatGPT the notion of SATD before querying it with any of the three above-listed prompts. However, we did not observe significant changes in the output, thus indicating that ChatGPT is “aware” of what SATD is.

D. Data Collection and Analysis

We run each trained CodeT5 on the 1,000 *Java* methods in the test set, asking it to implement the code changes needed to repay the SATD. We use the beam search decoding schema [71] to produce multiple candidate repayments for an input m_{satd} . In the case of ChatGPT, we use the OpenAI APIs to query it. However, the APIs do only allow collecting a single answer (solution) from ChatGPT.

In the following, we summarize the seven different models’ configurations we experiment with:

- 1: *No pre-training + context-specific fine-tuning*, in the results referred as M0;
- 1: *CodeT5 + context-specific fine-tuning*, referred as M1;
- 1: *CodeT5 + supervised pre-training on code changes + context-specific fine-tuning*, referred as M2_{CC};
- 1: *CodeT5 + supervised pre-training on code changes + no-context fine-tuning*, referred as M3_{CC-Ablation};
- 3: *ChatGPT in zero-shot learning setting* \times 3 prompt templates (M_{4T1-T3}), where the digit 1-3 indicates the used template among those described in Section III-C4.

We assess the performance of each model using two metrics. First, the percentage of Exact Match predictions for different beam sizes K (EM@K), namely the cases in which the

generated output is identical to the expected m_{fixed} . For CodeT5 we experiment with K equal 1, 3, 5, and 10. For the reasons previously explained, we only computed EM@1 for ChatGPT. Second, we compute the CrystalBLEU score [72] between the generated predictions and the m_{fixed} target. CrystalBLEU measures the similarity between a candidate (predicted code) and a reference code (oracle), similar to how the BLEU score [73] measures similarity between texts. However, CrystalBLEU is specifically designed for code evaluation, while retaining desirable properties of BLEU, specifically being language-agnostic and minimizing the effect of trivially shared n -grams, which would produce inflated results.

To better understand the extent to which the considered techniques can successfully address SATD, we analyze the edit actions (i.e., deleting, adding, moving, or changing) to code elements required in each SATD repayment instance. To this aim, we use the Gumtree Spoon AST Diff [74] to gather the Delete, Insert, Move, and Update actions performed on the source code AST nodes when SATD is being addressed. Specifically, we compute the *actual* AST edit actions, i.e., those obtained by differencing the input and the target (i.e., ground truth) of the model.

Subsequently, we create two separate buckets. The first bucket includes all methods where the best-performing model accurately addresses the SATD comment, while the second bucket includes methods for which the suggested code is inconsistent with the developer’s proposed repayment (i.e., ground truth). For both categories, we present the relative counts of AST edit actions learned by the model (when the code suggested for addressing the SATD is actually correct) and those where the model faces difficulties in providing a significant implementation.

Also, we perform statistical tests to determine whether one of the experimented techniques is more effective in producing code changes to address SATD. We use McNemar’s test [75] (with is a proportion test for dependent samples) and Odds Ratios (ORs) on the EMs that the techniques generate. We also statistically compare the distribution of the CrystalBLEU scores (computed at the sentence level) for the predictions generated by each technique by using the Wilcoxon signed-rank test [76]. The Cliff’s Delta (d) is used as effect size [77] and it is considered: negligible for $|d| < 0.10$, small for $0.10 \leq |d| < 0.33$, medium for $0.33 \leq |d| < 0.474$, and large for $|d| \geq 0.474$. For all tests, we assume a significance level of 95% and we account for multiple tests by adjusting p -values using Holm’s correction procedure [78].

Finally, we discuss examples of successfully addressed SATD comments by the top-performing model and, at the same time, we present cases where the model was unable to pay back the TD.

IV. RESULTS

Table II reports the results obtained by the studied techniques when addressing SATD instances from our test set. The first column (“Model”) provides a unique identifier we assigned to each of the 7 experimented techniques described in

TABLE II: Exact Match (*i.e.*, the recommended code is equal to the oracle) and CrystalBleu scores achieved by the different techniques when addressing SATD comments. In bold we report the highest value for both metrics when producing $K=1$, $K=3$, $K=5$, and $K=10$ candidate removals.

Model	Training Configuration				Top-1		Top-3		Top-5		Top-10	
	Self-supervised PT	Supervised PT	SATD Comm.	FT	EM	CB	EM	CB	EM	CB	EM	CB
M0	✗	✗	✓	✓	0%	22.13%	0%	25.98%	0.0%	25.43%	0.0%	25.14%
M1	✓	✗	✓	✓	2.23%	73.11%	5.40%	73.95%	6.10%	74.19%	7.20%	73.87%
M2 _{CC}	✓	✓	✓	✓	2.30%	73.41%	5.60%	74.20%	6.70%	74.28%	8.10%	74.25%
M3 _{CC-Ablation}	✓	✓	✗	✓	0.9%	72.58%	2.90%	73.48%	3.80%	73.60%	5.10%	73.50%
M4 _{T1}	✓	✗	✓	✗	1.18%	37.30%	-	-	-	-	-	-
M4 _{T2}	✓	✗	✓	✗	1.19%	49.68%	-	-	-	-	-	-
M4 _{T3}	✓	✗	✓	✗	0.0%	1.70%	-	-	-	-	-	-

TABLE III: Comparison among different techniques for top-1 predictions: McNemar’s and Wilcoxon’s test results.

Comparison	McNemar’s Test		Wilcoxon’s Test	
	<i>p</i> -value	OR	<i>p</i> -value	<i>d</i>
M1 vs. M0	-	-	<0.05	-0.86 (L)
M1 vs. M2 _{CC}	>0.05	1.0	>0.05	-0.01 (N)
M3 _{CC-Ablation} vs. M2 _{CC}	<0.05	8.0	<0.05	-0.01 (N)
M4 _{T1} vs. M2	>0.05	1.38	<0.05	-0.54 (L)
M4 _{T2} vs. M2	>0.05	1.36	<0.05	-0.43 (S)
M4 _{T3} vs. M2	-	-	<0.05	-0.98 (L)

Section III-D. The “Self-supervised PT” and “Supervised PT” indicates whether a specific configuration we tested featured the two types of pre-training, where the self-supervised is the one adopting the *masked language model* objective and the supervised uses the *generic code changes* dataset to provide the model with knowledge about changing code. The “SATD Comm.” column indicates whether the fine-tuning (or the prompting in the case of ChatGPT) included the SATD comment in the model’s input, while the “FT” column shows which model has been fine-tuned on our “SATD removal” dataset. Lastly, EM and CB indicate the performance of a specific configuration in terms of (i) the percentage of predictions that are Exact Matches (EM), and (ii) the average CrystalBLEU score (CB) [72] for all predictions in the test set. We present the results for different beam sizes (K) of 1, 3, 5, and 10.

Table III reports the results of the statistical tests (McNemar’s test and Wilcoxon signed-rank test), with adjusted p -values, OR, and Cliff’s d effect size. An $OR > 1$, or a positive Cliff’s d indicate that the right-side treatment outperforms the left-side one. To enhance readability, when doing the comparisons, we arranged the treatments to display $ORs \geq 1$.

A. RQ_1 : To what extent do pre-trained models of code support the automated SATD repayment?

The first two rows of Table II report the outcomes of the non-pre-trained model (M0) and its counterpart using self-supervised per-training on code and technical language (M1). While M0 is unable to address any SATD (EM = 0.0%) for all values of K , the prediction performances of M1 range from 2.23% ($K=1$) to 7.20% ($K=10$). M1 is a CodeT5 fine-tuned for SATD removal and our results stress the importance (and validity) of the pre-training procedure performed on it by the original authors [28]. Also, the difference in CrystalBLEU with respect to the non-pre-trained model (M0) is statistically significant (p -value < 0.05), according to Wilcoxon signed-

rank test, and is accompanied by a *Large* Cliff’s Delta. In the absence of EMs for M0, McNemar’s test results cannot be computed.

Answer to RQ_1 . The use of a self-supervised pre-trained model (CodeT5) has a significantly positive benefit when addressing SATD, if compared to a non-pre-trained model. The latter is unable to produce exact matches, likely due to the limited size of the fine-tuning dataset.

B. RQ_2 : To what extent does the infusion of “similar-task knowledge” in pre-trained models of code benefits the automated SATD repayment?

Instilling task-similar (*i.e.*, code changes) knowledge into the model (M2_{CC}, featuring both self-supervised and supervised pre-training) results in a slight performance improvement as compared to M1 (*i.e.*, self-supervised pre-training only). While there is an improvement across all beam sizes (see Table II), this is usually minor. For example, when only relying on the top prediction (*i.e.*, $K=1$), the EMs rise from 2.23% to 2.30% which, given the 1,000 instances featured in our test set, means 7 new EM predictions. The improvement is slightly higher when looking at higher values of K , with a +0.9% reached for $K=10$ (7.20% vs. 8.10%). Upon statistically comparing both models (M1 vs. M2_{CC}), the McNemar’s test (Table III) reports a lack of significant differences (p -value > 0.05) in EM predictions between M1 and M2_{CC}. Wilcoxon signed-rank test also suggests non-significant differences in the distributions of CrystalBLEU scores between M1 and M2_{CC}. Such a result is in line with what was observed by Tufano *et al.* [79]. They found that adopting a pre-training objective resembling the downstream task does not always substantially help, questioning the effort needed for the additional training time. This also seems to be the case when addressing SATD. Despite this, M2_{CC} still is the best-performing model we experimented with and, for this reason, we performed some additional analyses on its predictions.

The EM predictions generated by M2_{CC} with $K=10$, feature a total of 768 AST *edit actions* correctly implemented by the model. Out of these, 62.36% are Delete operations (*i.e.*, an AST node is removed), 33.60% are Inserts (*i.e.*, a new node is introduced into the AST), and 2.34% and 1.70% are Move and an Update operations, respectively

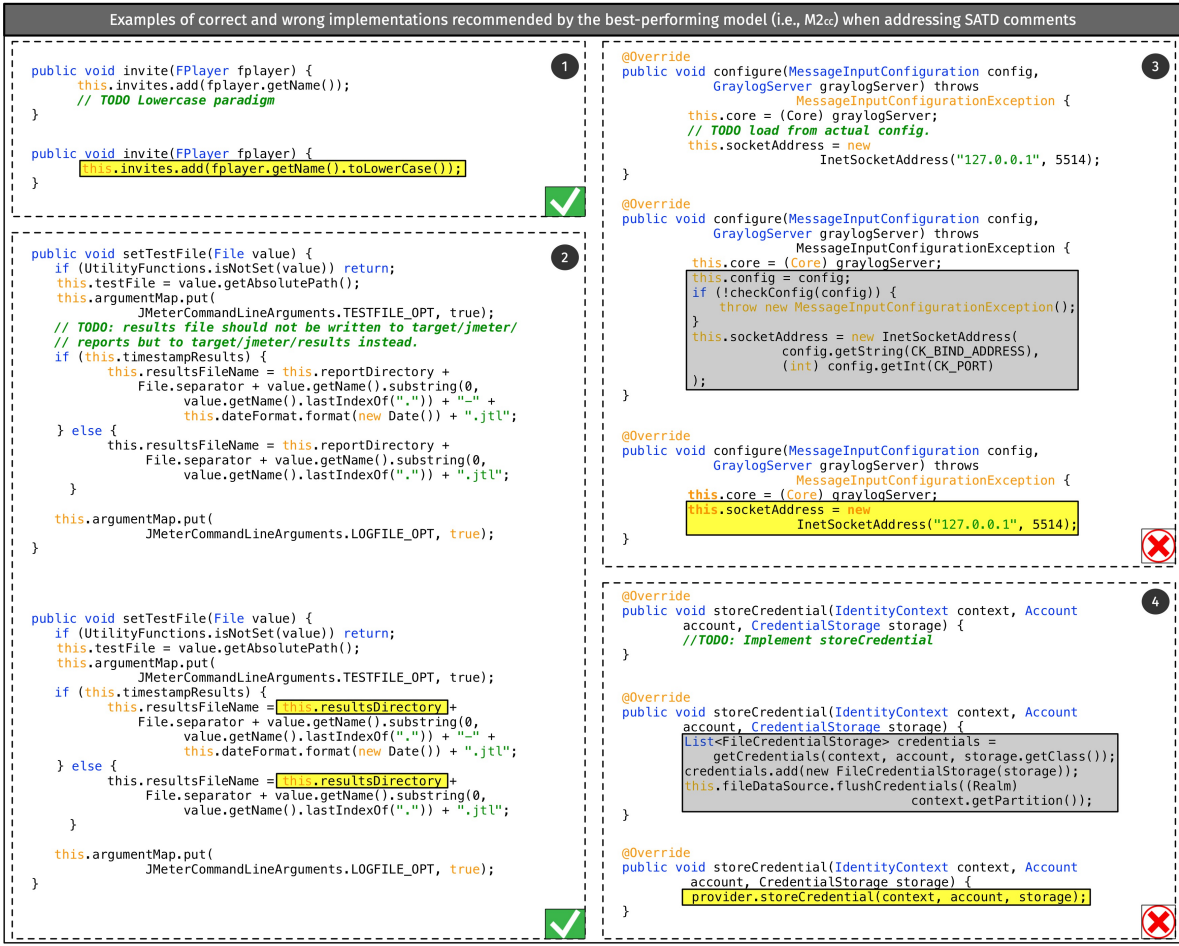


Fig. 3: Example of 4 SATD comments addressed by the model (2 correct and 2 wrong) for top-10 candidate recommendations.

When looking at the failure cases (i.e., non-EMs), the distribution of AST *edit actions* that was needed to address the SATD (but that the model failed to reproduce) we found that out of the EMs: 32.01% are Deletions, 54.10% Insertions, 6.84% Moves, and 7.05% Updates . Thus, are not the “types” of AST edits needed to address the SATD that discriminate what the model can or cannot do. Given this finding, we also computed the sheer number of AST *edit actions* that were needed in EMs and wrong predictions to address the SATD. The achieved results, as expected, indicate that the model struggles in addressing SATD in need of a high number of AST changes to be repaid. Indeed, the median number of changes that were required to address the SATD instances that resulted in EMs is 6, as compared to the 20 of the wrong predictions (mean 9.8 vs 38.1).

Fig. 3 illustrates four instances from our test set, including two for which M2_{CC} was able to successfully address the SATD (1 and 2), and two for which it failed to pay back the TD (3 and 4). For the successful cases, the code on top shows the input method including the SATD, while the one at the bottom shows how the model addressed the SATD (changes highlighted by the yellow boxes). For the failure cases, we also report the expected target from our dataset, namely the code showing how the developers actually

addressed the SATD (changes highlighted with grey boxes).

In 1 the SATD mentions “TODO Lowercase paradigm”. To fulfill this requirement, the model performs two distinct AST edit actions, addressing the TD in the right location by injecting the `toLowerCase` invocation. The example of scenario 2 shows how the model addresses the SATD by replacing the `reportDirectory` attribute in the current instance with `resultsDirectory` through a code change that involves updating two AST nodes (i.e., an Update edit) in the *if* and *else* statements.

In scenario 3 the model fails to effectively handle the SATD comment (TODO load from actual config) by requiring modifications to the instantiation of `this.socketAddress`. The hard-coded IP address and port need to be replaced with values fetched using the `config` method. The implementation of these changes is unsuccessful, as the model outputs the same method (M_{sadt}) with the SATD comment removed (gray box in 3). There are a total of 21 AST edit actions to be implemented to successfully address the SATD comment, including 18 Insert operations, two Delete operations, and one Update action. When considering scenario 4, the SATD left by the developer requires the complete implementation of the `storeCredential` method. Nonetheless, the recommendation provided by the

model does not address the SATD comment appropriately since it assumes the existence of a *storeCredential* method that takes *context*, *account*, and *storage* as input parameters, failing to address the TD. A successful change would have required the addition of 19 new nodes (*i.e.*, Insert) to the AST of the *Java* method *storeCredential*.

Answer to RQ₂. Seeding task-similar knowledge (*e.g.*, code changes) into a model pre-trained using self-supervised task models of code only slightly improves their performance. Even our best-performing model struggles in addressing SATD instances requiring a large number of AST edit actions.

C. RQ₃: To what extent does the presence of “context-specific knowledge” help pre-trained models of code in the automated SATD repayment?

By comparing the results in row 4 of Table II ($M_{CC-Ablation}$) with those in row 3 (M_{CC}), we can observe the fundamental role played by the context-specific knowledge provided as input to the model (*i.e.*, the SATD comment) in automatically addressing TD. Admitting TD through a comment aids the model to better perform for all considered beam sizes (K). For example, when focusing on a single candidate solution (*i.e.*, top-1), $M_{CC-Ablation}$ can only achieve an EM in 0.9% of cases, while M_{CC} does it in 2.30% of cases. When looking at higher values of K , the gap becomes even larger, up to a +3% for $K = 10$ (5.10% vs. 8.10%). McNemar’s test indicates a significant difference (p -value < 0.05) between M_{CC} and $M_{CC-Ablation}$, with M_{CC} having 8 times higher odds ($OR=8$) to address SATD than $M_{CC-Ablation}$. Instead, although the differences found by Wilcoxon signed-rank test for the CrystalBLEU are statistically significant, the effect size is *negligible*. The obtained results further highlight the importance for developers to admit TD. In essence, SATD does not only serve as a trace for themselves and for other developers [3], but, also, as a way to better guide automated tools in addressing TD. Furthermore, this stresses the importance of recommending developers that TD should be admitted [36].

Answer to RQ₃. The availability of context-specific knowledge in the form of SATD comments enhances the performance of pre-trained models of code, allowing them to achieve a substantial increase in the percentage of automatically addressed TD. This is a further motivation for developers to admit TD in their source code.

D. RQ₄: Are general-purpose large language models zero-shot learners for SATD repayment?

The last three rows of Table II report the performances achieved by ChatGPT as *zero-shot learner* for addressing SATD. Two findings emerge: (i) the usage of different templates to prompt ChatGPT for the task of SATD repayment plays a crucial role and, (ii) the performances achieved when

recommending one single candidate solution (top-1) are lower than DL-based techniques appositely fine-tuned (M_1 and M_{2CC}) to pay back TD. As for the use of several prompt templates, it is important to note that designing templates showing the code first and the comment later (as did in M_{4T_3}) strongly penalize the model, resulting in 0 EM and the lowest CrystalBLEU score across all treatments. Differently, providing the SATD comment first and the code including such a comment later helps ChatGPT in achieving better performances, with 1.18% and 1.19% of SATD comments successfully addressed, respectively for M_{4T_1} and M_{4T_2} .

McNemar’s test on the top-1 recommendation indicates no significant differences (p -value > 0.05) between M_{4T_1} and M_{2CC} , as well as M_{4T_2} and M_{2CC} . However, there are statistically significant differences when comparing the CrystalBLEU distributions of the tested templates with M_{2CC} . In these instances, the effect is *large* for M_{4T_1} and M_{4T_3} , and small for M_{4T_2} , where ChatGPT performed best.

Without knowing the details of ChatGPT’s implementation, it is hard to speculate on the reasons behind such performances. Possibly, they could be related to the lack of a specific fine-tuning for the specific task, or also to the need to generate suitable prompts.

Answer to RQ₄. When used in a zero-shot setting, LLMs, and ChatGPT in particular, exhibit sub-optimal performance compared to pre-trained models of code appositely tuned for the specific task of addressing SATD. Additionally, the use of well-crafted templates plays a crucial role for LLMs being used off-the-shelf.

V. THREATS TO VALIDITY

Construct validity. The main issue to be considered is whether the observed SATD removals are true positives. To mitigate this threat, we only considered cases in which the SATD comment contained well-recognized keywords, and we manually analyzed a sample to check for problematic cases.

To avoid training our model on instances where the SATD was removed by chance, we excluded cases where the affected method was removed. We are, however, aware that the latter heuristics, while mitigating some threats to construct validity, could affect the study’s generalizability.

Internal validity. As explained in Section III-C, we used the default settings of the employed language models. Better results could be achieved with proper hyperparameter tuning. The assumption we have made in Section III-B which links the removal of an SATD comment with the change performed within the same method is subject to imprecision. First, the commit could tangle the SATD repayment with other changes. Second, the comment removal may be out of sync with the source code change aimed at addressing the SATD.

Conclusion validity. The comparison between different techniques is supported by suitable statistical procedures and effect size measures. Also, the results of multiple tests have

been adjusted through Holm’s procedure [78]. We are aware that the performance of the studied approaches may change, and possibly improve, if experimenting with a larger dataset.

External validity. Our study is limited in terms of programming language (Java) and, more important, the SATD removal dataset is based on 1,000 instances. As stated in Section III-C, we limit to addressing SATD that have been resolved within the same method, and to methods not longer than 1024 tokens. As the paper aims to set—within the employed generative models—an “upper bound” of the SATD repayment capability, we do not expect any better results for more complex and extensive changes. Better generalizability of our results would require studies on further and more diversified datasets. In terms of considered models, our results are limited to the CodeT5-base [28], as well as a zero-shot attempt done with ChatGPT [32]. Other models having a different size and architecture could, possibly, exhibit different results. However, in this circumstance, our interest was to mainly show the feasibility of SATD removal and, within the same model (CodeT5 in our case), the relative improvements with different levels of pre-training and fine-tuning. Moreover, we acknowledge that we have not experimented with edit-specific models, such as CoditT5 [53], and therefore further experimentation with such models would be desirable. At least, we partially mitigated this threat by experimenting in RQ₂ a pre-training with code changes. Moreover, as explained in Section II-C, our dataset mostly features removals and additions rather than updates and moves.

Last, but not least, also the preliminary results with ChatGPT need to be confirmed or confuted with similar yet differently implemented tools, *e.g.*, Google Bard [80].

VI. DISCUSSION AND CONCLUSION

In this paper, we investigated the use of Deep Learning (DL) models for automatically addressing technical debt (TD). To train the models, we leveraged 5,039 instances of SATD removals mined using an existing tool [27]. Such a small number of training instances made the pre-training of the model absolutely necessary to achieve an automated fixing of SATD instances. Nevertheless, even the best-performing model we experimented with can automatically repay SATD only in a minority of cases (2% to 8%). The complexity of such a task has also been confirmed by the results achieved exploiting the state-of-the-art LLM (*i.e.*, ChatGPT [32]), that under-performed if compared to the specialized models we tested. So far, the use of generative deep learning models has been successful for several code tasks that require either generation of new code, or the change of existing one. Some (non-exhaustive) examples of achieved performances are on the order of $\sim 14\%$ for bug fixing (Wang *et al.* [28]), of $\sim 5\%$ for generating a reviewed version of existing source code (Tufano *et al.* [23]), and $\sim 23\%$ for generating code blocks (Ciniselli *et al.* [12]). As far as SATD repayment is concerned, we have observed both positive results and negative results. On the positive side, results are in line with some existing approaches for automated bug fixing [81].

On the negative side:

- 1) The level of performances achieved so far would still limit the applicability of the proposed approach in real practice. The latter may also possibly require some explanation/rationale of the changes to be performed, *e.g.*, telling to the developer that the change being done is enacting a refactoring action, fixing a bug, improving the code readability, etc.
- 2) Such results have been obtained under the assumption that the change concern a single method, with a limited maximum size of the considered methods, and assuming that the removal of a SATD comment would correspond to changes aimed at addressing it.
- 3) Although the proposed approach was able to correctly recommend instances of SATD repayment involving the addition of an AST (over 33% of our exact matches), the approach may fail to introduce a totally new, unseen piece of feature, as well as it is unlikely to be able to perform changes such as API upgrade/replacements.

The aforementioned limitations greatly stimulate future research in this area. First and foremost, although in RQ₂ we have performed a large training of CodeT5 on a dataset of code changes, it would be worthwhile to experiment with models more specifically suited for code edits, such as CoditT5 [53].

Second, although RQ₄ has shown that a zero-shot instance of ChatGPT fails to support the SATD repayment task, other investigations with LLMs are worthwhile, as those might be able to help in supporting larger change tasks. They can go in the direction of prompt engineering, as well as experimenting with other LLMs. Third, given the variety of the SATD nature, it may be worthwhile to pursue the development of eclectic approaches, that first classify the type of needed change (*e.g.*, along the line of what SARDELE [10] does) and then employing different models or even different approaches for each of them. Our future work targets (i) the possibility to exploit LLMs in a few-shot learning scenario (rather than the zero-shot we experimented with), and (ii) the definition of different mining pipelines which could help enlarging the SATD removal dataset, possibly boosting performance.

VII. DATA AVAILABILITY

Our replication package [33] includes all code and data employed in our research. This comprises the datasets needed for training and testing the models, the code to reproduce the experiments (*i.e.*, models’ training and inference), the predictions generated in each setting, and the R scripts employed for performing statistical analysis.

VIII. ACKNOWLEDGMENTS

This project has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement No. 851720). Di Penta acknowledges the Italian “PRIN 2021” project EMELIOT “Engineered Machine Learning-intensive IoT systems.”

REFERENCES

- [1] W. Cunningham, “The wycash portfolio management system,” in *Addendum to the Proceedings on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA 1992 Addendum, Vancouver, British Columbia, Canada, October 18-22, 1992*. ACM, 1992, pp. 29–30.
- [2] A. Potdar and E. Shihab, “An exploratory study on self-admitted technical debt,” in *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*. IEEE Computer Society, 2014, pp. 91–100.
- [3] F. Zampetti, G. Fucci, A. Serebrenik, and M. Di Penta, “Self-admitted technical debt practices: a comparison between industry and open-source,” *Empir. Softw. Eng.*, vol. 26, no. 6, p. 131, 2021. [Online]. Available: <https://doi.org/10.1007/s10664-021-10031-3>
- [4] E. da S. Maldonado, R. Abdalkareem, E. Shihab, and A. Serebrenik, “An empirical study on the removal of self-admitted technical debt,” in *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*, 2017, pp. 238–248.
- [5] F. Zampetti, A. Serebrenik, and M. Di Penta, “Was self-admitted technical debt removal a real removal?: an in-depth perspective,” in *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*. ACM, 2018, pp. 526–536.
- [6] G. Bavota and B. Russo, “A large-scale empirical study on self-admitted technical debt,” in *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*, 2016, pp. 315–326.
- [7] G. Fucci, N. Cassee, F. Zampetti, N. Novielli, A. Serebrenik, and M. Di Penta, “Waiting around or job half-done? sentiment in self-admitted technical debt,” in *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021, Madrid, Spain, May 17-19, 2021*, 2021, pp. 403–414. [Online]. Available: <https://doi.org/10.1109/MSR52588.2021.00052>
- [8] E. da S. Maldonado, E. Shihab, and N. Tsantalis, “Using natural language processing to automatically detect self-admitted technical debt,” *IEEE Trans. Software Eng.*, vol. 43, no. 11, pp. 1044–1062, 2017.
- [9] X. Ren, Z. Xing, X. Xia, D. Lo, X. Wang, and J. Grundy, “Neural network-based detection of self-admitted technical debt: From performance to explainability,” *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 3, p. 15, 2019.
- [10] F. Zampetti, A. Serebrenik, and M. Di Penta, “Automatically learning patterns for self-admitted technical debt removal,” in *27th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2020, London, ON, Canada, February 18-21, 2020*. IEEE, 2020, pp. 355–366.
- [11] M. Ciniselli, N. Cooper, L. Pascarella, D. Shpyvanyk, M. Di Penta, and G. Bavota, “An empirical study on the usage of bert models for code completion,” in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 108–119.
- [12] M. Ciniselli, N. Cooper, L. Pascarella, A. Mastropaolo, E. Aghajani, D. Shpyvanyk, M. Di Penta, and G. Bavota, “An empirical study on the usage of transformer models for code completion,” *IEEE Trans. Software Eng.*, vol. 48, no. 12, pp. 4818–4837, 2022. [Online]. Available: <https://doi.org/10.1109/TSE.2021.3128234>
- [13] A. Svyatkovskiy, S. Lee, A. Hadjitofi, M. Riechert, J. Franco, and M. Allamanis, “Fast and memory-efficient neural code completion,” 2020.
- [14] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, “Intellicode compose: Code generation using transformer,” in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1433–1443.
- [15] J. Li, Y. Wang, M. R. Lyu, and I. King, “Code completion with neural attention and pointer networks,” *arXiv preprint arXiv:1711.09573*, 2017.
- [16] E. Mashhadi and H. Hemmati, “Applying codebert for automated program repair of java simple bugs,” in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 2021, pp. 505–509.
- [17] N. Jiang, T. Lutellier, and L. Tan, “Cure: Code-aware neural machine translation for automatic program repair,” in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1161–1173.
- [18] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, “Coconut: combining context-aware neural translation models using ensemble for program repair,” in *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, 2020, pp. 101–114.
- [19] Y. Li, S. Wang, and T. N. Nguyen, “Dlfix: Context-based code transformation learning for automated program repair.” New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: <https://doi.org/10.1145/3377811.3380345>
- [20] Z. Chen, S. Kommrusch, and M. Monperrus, “Neural transfer learning for repairing security vulnerabilities in c code,” *IEEE Transactions on Software Engineering*, vol. 49, no. 1, pp. 147–165, 2022.
- [21] M. Fu, C. Tantithamthavorn, T. Le, V. Nguyen, and D. Phung, “Vulrepair: a t5-based automated software vulnerability repair,” in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 935–947.
- [22] R. Tufano, L. Pascarella, M. Tufano, D. Shpyvanyk, and G. Bavota, “Towards automating code review activities,” in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 163–174.
- [23] R. Tufano, S. Masiero, A. Mastropaolo, L. Pascarella, D. Shpyvanyk, and G. Bavota, “Using pre-trained models to boost code review automation,” in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, 2022*, pp. 2291–2302.
- [24] Z. Li, S. Lu, D. Guo, N. Duan, S. Jannu, G. Jenks, D. Majumder, J. Green, A. Svyatkovskiy, S. Fu *et al.*, “Codereviewer: Pre-training for automating code review activities,” *arXiv preprint arXiv:2203.09095*, 2022.
- [25] —, “Automating code review activities by large-scale pre-training,” in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 1035–1047.
- [26] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention is all you need,” *Advances in neural information processing systems*, vol. 30, 2017.
- [27] E. A. AlOmar, B. Christians, M. Busho, A. H. AlKhalid, A. Ouni, C. Newman, and M. W. Mkaouer, “Satdbailiff-mining and tracking self-admitted technical debt,” *Science of Computer Programming*, vol. 213, p. 102693, 2022.
- [28] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, “Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation,” *arXiv preprint arXiv:2109.00859*, 2021.
- [29] C. Wang, Y. Yang, C. Gao, Y. Peng, H. Zhang, and M. R. Lyu, “No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence,” in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 382–394.
- [30] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” 2018.
- [31] Y. Liu, M. Ott, N. Goyal, J. Du, N. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, “Roberta: A robustly optimized bert pretraining approach,” *arXiv preprint arXiv:1907.11692*, 2019.
- [32] “Chatgpt <https://openai.com/blog/chatgpt>.”
- [33] “Replication package <https://github.com/antonio-mastropaolo/SATD-Removal>.”
- [34] C. Watson, N. Cooper, D. N. Palacio, K. Moran, and D. Shpyvanyk, “A systematic literature review on the use of deep learning in software engineering research,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 2, pp. 1–58, 2022.
- [35] N. Cassee, F. Zampetti, N. Novielli, A. Serebrenik, and M. Di Penta, “Self-admitted technical debt and comments’ polarity: an empirical study,” *Empir. Softw. Eng.*, vol. 27, no. 6, p. 139, 2022.
- [36] F. Zampetti, C. Noiseux, G. Antoniol, F. Khomh, and M. Di Penta, “Recommending when design technical debt should be self-admitted,” in *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*, 2017, pp. 216–226.
- [37] G. Sierra, E. Shihab, and Y. Kamei, “A survey of self-admitted technical debt,” *Journal of Systems and Software*, vol. 152, pp. 70–82, 2019.
- [38] J. Tan, D. Feitosa, and P. Avgeriou, “Do practitioners intentionally self-fix technical debt and why?” in *IEEE International Conference on Software Maintenance and Evolution, ICSME 2021, Luxembourg, September 27 - October 1, 2021*. IEEE, 2021, pp. 251–262.
- [39] D. Pina, C. Seaman, and A. Goldman, “Technical debt prioritization: a developer’s perspective,” in *TechDebt@ICSE*. ACM, 2022, pp. 46–55.

- [40] Z. Chen, S. Komrusch, M. Tufano, L.-N. Pouchet, D. Poshyanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1943–1959, 2019.
- [41] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 4, pp. 19:1–19:29, 2019.
- [42] M. Tufano, J. Pantuchina, C. Watson, G. Bavota, and D. Poshyanyk, "On learning meaningful code changes via neural machine translation," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 25–36.
- [43] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever *et al.*, "Improving language understanding by generative pre-training," 2018.
- [44] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.
- [45] R.-M. Karampatsis and C. Sutton, "How often do single-statement bugs occur? the manystubs4j dataset," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 573–577.
- [46] A. Mastropaolo, S. Scalabrino, N. Cooper, D. N. Palacio, D. Poshyanyk, R. Oliveto, and G. Bavota, "Studying the usage of text-to-text transfer transformer to support code-related tasks," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 336–347.
- [47] A. Mastropaolo, N. Cooper, D. N. Palacio, S. Scalabrino, D. Poshyanyk, R. Oliveto, and G. Bavota, "Using transfer learning for code-related tasks," *IEEE Transactions on Software Engineering*, 2022.
- [48] T. Kudo and J. Richardson, "Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing," *arXiv preprint arXiv:1808.06226*, 2018.
- [49] P. Thongtanunam, C. Pornprasit, and C. Tantithamthavorn, "Autotransform: Automated code transformation to support modern code review process," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 237–248.
- [50] S. Brody, U. Alon, and E. Yahav, "A structural model for contextual code changes," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 215:1–215:28, 2020.
- [51] Z. Chen, V. J. Hellendoorn, P. Lamblin, P. Maniatis, P. Manzagol, D. Tarlow, and S. Moitra, "PLUR: A unifying, graph-based view of program learning, understanding, and repair," in *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, 2021, pp. 23 089–23 101.
- [52] Z. Yao, F. F. Xu, P. Yin, H. Sun, and G. Neubig, "Learning structural edits via incremental tree transformations," in *ICLR*. OpenReview.net, 2021.
- [53] J. Zhang, S. Panthaplackel, P. Nie, J. J. Li, and M. Gligoric, "CoditT5: pretraining for source code and natural language editing," in *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 2022, pp. 22:1–22:12.
- [54] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.
- [55] OpenAI, "Gpt-4 technical report," 2023.
- [56] J. A. Prenner and R. Robbes, "Automatic program repair with openai's codex: Evaluating quixbugs," *arXiv preprint arXiv:2111.03922*, 2021.
- [57] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.
- [58] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *The Journal of Machine Learning Research*, vol. 21, no. 1, pp. 5485–5551, 2020.
- [59] S. Zhou, U. Alon, F. F. Xu, Z. Jiang, and G. Neubig, "Docprompting: Generating code by retrieving the docs," in *The Eleventh International Conference on Learning Representations*, 2022.
- [60] K. Kusum, A. Ahmed, C. Bhuvana, and V. Vivek, "Unsupervised translation of programming language-a survey paper," in *2022 4th International Conference on Advances in Computing, Communication Control and Networking (ICAC3N)*. IEEE, 2022, pp. 384–388.
- [61] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," *arXiv preprint arXiv:2103.06333*, 2021.
- [62] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020.
- [63] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "CodeSearchNet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.
- [64] "Gerrit." [Online]. Available: <https://www.gerritcodereview.com>
- [65] A. Mastropaolo, L. Pascarella, and G. Bavota, "Using deep learning to generate complete log statements," *arXiv preprint arXiv:2201.04837*, 2022.
- [66] M. Tufano, D. Drain, A. Svyatkovskiy, S. K. Deng, and N. Sundaresan, "Unit test case generation with transformers," *CoRR*, vol. abs/2009.05617, 2020. [Online]. Available: <https://arxiv.org/abs/2009.05617>
- [67] O. Dabic, E. Aghajani, and G. Bavota, "Sampling projects in github for msr studies," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 560–564.
- [68] H. W. Cedric Richter, "Tssb-3m: Mining single statement bugs at massive scale," in *MSR*, 2022.
- [69] J. Zhang, Y. Zhao, M. Saleh, and P. Liu, "Pegasus: Pre-training with extracted gap-sentences for abstractive summarization," in *International Conference on Machine Learning*. PMLR, 2020, pp. 11 328–11 339.
- [70] B. Wei, G. Li, X. Xia, Z. Fu, and Z. Jin, "Code generation as a dual task of code summarization," *Advances in neural information processing systems*, vol. 32, 2019.
- [71] M. Freitag and Y. Al-Onaizan, "Beam search strategies for neural machine translation," *arXiv preprint arXiv:1702.01806*, 2017.
- [72] A. Eghbali and M. Pradel, "Crystalbleu: precisely and efficiently measuring the similarity of code," in *37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.
- [73] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.
- [74] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*. ACM, 2014, pp. 313–324.
- [75] Q. McNemar, "Note on the sampling error of the difference between correlated proportions or percentages," *Psychometrika*, vol. 12, no. 2, pp. 153–157, 1947.
- [76] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics Bulletin*, vol. 1, no. 6, pp. 80–83, 1945.
- [77] R. J. Grissom and J. J. Kim, *Effect sizes for research: A broad practical approach*, 2nd ed. Lawrence Earlbaum Associates, 2005.
- [78] S. Holm, "A simple sequentially rejective multiple test procedure," *Scandinavian journal of statistics*, pp. 65–70, 1979.
- [79] R. Tufano, L. Pascarella, and G. Bavota, "Automating code-related tasks through transformers: The impact of pre-training," in *45th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2023*, 2023, p. To appear.
- [80] I. Google, "Try Bard, an AI expertiment by Google <https://bard.google.com>," 2023.
- [81] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 4, pp. 1–29, 2019.