Using Deep Learning to Automatically Improve Code Readability

Antonio Vitale, Valentina Piantadosi, Simone Scalabrino, Rocco Oliveto
STAKE Lab @ University of Molise, Italy

***Abstract*—Reading source code occupies most of developer's daily activities. Any maintenance and evolution task requires developers to read and understand the code they are going to modify. For this reason, previous research focused on the definition of techniques to automatically assess the readability of a given snippet. However, when many unreadable code sections are detected, developers might be required to manually modify them all to improve their readability. While existing approaches aim at solving specific readability-related issues, such as improving variable names or fixing styling issues, there is still no approach to automatically suggest which actions should be taken to improve code readability.**

In this paper, we define the first holistic readability-improving approach. As a first contribution, we introduce a methodology for automatically identifying readability-improving commits, and we use it to build a large dataset of 122k commits by mining the whole revision history of all the projects hosted on GitHub between 2015 and 2022. We show that such a methodology has ∼86% accuracy. As a second contribution, we train and test the T5 model to emulate what developers did to improve readability. We show that our model achieves a perfect prediction accuracy between 21% and 28%. The results of a manual evaluation we performed on 500 predictions shows that when the model does not change the behavior of the input and it applies changes (34% of the cases), in the large majority of the cases (79.4%) it allows to improve code readability.

***Index Terms*—code readability, large language models, t5**

I. Introduction

Software developers spend much of their time reading and understanding code. In fact, the majority of software evolution and maintenance time (and, therefore, budget) is devoted to such an activity [10], [33]. Unfortunately, however, some software projects contain a large quantity of unreadable code [37]. Given the relevance of the problem, previous work aimed at defining models to automatically assess the code readability [4], [5], [9], [32], [38], [43], [44]. While detecting unreadable code is beneficial to developers, the problem of having unreadable code remains, unless someone improves such artifacts. Developers, indeed, are left with a hard choice: Taking the burden of manually improving the readability of unreadable code, spending precious time in doing so, or keeping it as is, accruing technical debt that they will inevitably pay in the future. Automatically improving code readability might be highly beneficial to developers since it would allow them to reduce the technical debt of their software systems at a lower cost. Several tools are available to help developers in this task. For example, IDEs provide formatters that automatically adjust the indentation and automatically fix the most basic formatting errors (*e.g.,* whitespace-related ones). Also, previous work presented approaches aimed at improving aspects naturally related to code readability, like styling [25] and naming [1], [24], [46]. Still, to the best of our knowledge, no previous work introduced an approach specifically aimed at improving code readability as a whole.

Ideally, such an approach would take as input a snippet s with readability-related issues and return a new version of s (s^*) without such issues. A viable solution is using Deep Learning (DL) and, specifically, Large Language Models (LLMs). LLMs have been successfully adopted for several coding task, like automated bug fixing and generation of assertion statements for test cases [28]. To use such approaches, however, plenty of pairs (s, s^*) are needed to fine tune the model. Manually defining an adequately large dataset is not an option because of the colossal amount of work required.

In this regard, software repositories might be a valuable source of information. Developers improve code readability in the evolution of open-source software projects all the time. Let us consider as an example commit `3acfc16` from the GitHub repository `RoboJackets/roboracing-software` [41]. The commit message, "Made the code readable," clearly states the developer's intention. Automatically collecting such commits would allow us to fine tune a LLM able to automatically improve code readability. Still, this is not an easy task. A simple keyword-based approach would not work, because the word "readability" is used in a variety of contexts: It could be related not only to code, but also to the output formatting (*e.g.,* logging) or to GUI-related elements (*e.g.,* font readability).

In this paper, we define the first holistic DL-based approach for automatically improving code readability. Our approach does not require humans to manually improve the readability of code snippets. As a first contribution, we define a novel methodology for automatically selecting readability-improving commits through a set of heuristics, including a novel NLP-based filter. We use it to mine the whole history of GitHub between Jan 2015 and Dec 2022. As a result, we obtain 122k code-readability improvement operations. We manually analyzed a significant sample of such commits to validate our approach, and we found that 86% of the selected commits actually aim at improving code readability. As a second contribution, we use our dataset to fine-tune and test a LLM to automatically improve code readability. To do this, we fine-tune T5, a state-of-the-art LLM used for other coding tasks [28]. Our results show that our model achieves an accuracy level between 21% (if only a single prediction is provided) and 28% (when providing up to 50 alternatives to the developers).

We also manually investigated a significant sample of 500 predictions to understand if (i) the model does not change the behavior of the input code, and (ii) it actually improves code readability, regardless of the fact that it perfectly matches what developers did. We found that the model tries to improve code readability (*i.e.,* it does not change the behavior of the code and it modifies the input code) in 34% of the instances, mostly because for many non-perfect predictions (47%) it does not change the input code at all. When checking the capability of the model of actually improving readability when considering changes in which behavior does not change and the model modifies the input code, we found that 83% of perfect predictions and 75% of non-perfect predictions have a positive effect on readability. In summary, our results show that not only the model improves readability, but it is generally safe to use since it changes the behavior of the input code in a minority of the cases (31%).

II. Related Work

In this section, we report both previous studies on code readability and deep-learning approaches defined to improve code-related features.

A. Code Readability

Erlikh *et al.* [10] showed that developers spend more time in maintenance activities than on development activities. For this reason, previous work aimed at defining approaches for automatically assessing code readability. Buse and Weimer [4], [5] introduced the first approach for achieving this goal through a set of structural features (*e.g.,* line length and identifier length). Later, Posnett *et al.* [38] defined a simpler model for code readability by using only three metrics (Halstead volume, entropy, and LOC). Dorn [9] contributed by introducing a very large dataset of manually-evaluated snippets in three programming languages, and he presented a new model including visual, spatial, and linguistic features (*e.g.,* alignment of characters and indentation variation) that allow to achieve higher prediction accuracy. Scalabrino *et al.* [43], [44] introduced a set of textual features (*e.g.,* consistency between comments and identifiers and comment readability) and defined a comprehensive model including all the state-of-the-art features. Finally, Mi *et al.* [32] used for the first time deep-learning to predict code readability.

Readability prediction models, however, have been proved to have some limits in capturing readability improvements made by developers. Pantiuchina *et al.* [34] empirically investigated whether quality metrics were able to capture code quality improvement as perceived by developers. Authors performed an investigation on 1,282 commits with only Java files. They analyzed Java files to understand if the developers really improved code readability. Authors mined commits in which developers clearly stated their aim of improving code readability. Results demonstrated that more often the considered quality metrics were not able to capture the quality improvement as perceived by developers. Fakhoury *et al.* [11] consolidated the results obtained by Pantiuchina *et al.* [34].

Fakhoury *et al.* investigated ∼63 project with readability models and their results showed that readability models failed to capture readability improvements. Pantiuchina *et al.* [34] and Fakhoury *et al.* [11] introduced dataset including commits where developers explicitly stated readability improvements. Finally, Piantadosi *et al.* [37] showed that such models are much less accurate when used to predict if a commit changed the readability of a file. In all these studies, manual validation of the readability-improvement commits acquired was required because they used a simple keyword-matching approach to select readability-improving commits. As a result, their dataset contain few commits: The one by Pantiuchina *et al.* contain a total of 1,282 changes (including the ones related to other quality metrics), while the one by Fakhoury *et al.* contains 548 instances [11], and they both focused only on Java. Our approach for selecting readability-improving commits automatically discards possible false-positives identified through keyword-matching thanks to a NLP-based filter and other heuristics. In total, our dataset contains more than 122k readability-improving commits.

B. Deep Learning to Improve Code

In recent years, the use of deep learning (DL), thanks in part to the advent of LLMs (large language models), has become a very effective technique for coding tasks. Although there are plenty studies concerning the use of DL for coding tasks, we focus only on a few of them. A broader overview is available in the surveys by Allamanis *et al.* [2], Watson *et al.* [54], and Yang *et al.* [56].

One of the pioneering studies on the bug fixing through DL is the one of Tufano *et al.* [48]. The authors collected ∼787k commits, abstracted the buggy code and fixed code, and used them to train an Encoder-Decoder model able to translate buggy code into its fixed version founding that such a model was able to fix thousands of unique buggy methods.

Watson *et al.* [55] introduced *ATLAS* (Automatic Learning of Assert Statements), the first approach to generate assert statements in test methods. The authors mined 2.5M test methods from GitHub with their corresponding assert statement. The model was capable of successfully predicting over 31% of the assert statements developers wrote.

Raffel *et al.* [40] introduced T5 (*Text-To-Text Transfer Transformer*) in the field of Natural Language Processing. T5 allows to pre-train a model in a self-supervised way, and then to fine-tune it on the specific task. Mastropaolo *et al.* [28], [29] used T5 for coding tasks, and they showed that it allows to improve the state-of-the-art models for automated bug fixing [48], code summarization [14], assert generation [55], and mutant generation [47].

Ciniselli *et al.* [8] presented a large-scale study with the aim of exploring the capabilities of Transformers [50] on code completion of three different granularity levels: single tokens, one or multiple statements, and code blocks. They achieved a level of perfect predictions ranging from ∼29% (block prediction) to ∼69% (token prediction).

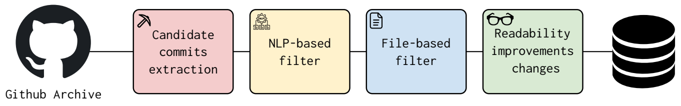
Github Archive
Candidate commits extraction
NLP-based filter
File-based filter
Readability improvements changes

Fig. 1: Process used for building the dataset.

Tufano *et al.* [49] investigated the possibility of automating specific code review tasks through the use of DL models targeting the automation of two tasks: (i) take in input a code submitted for review and implement in it changes likely to be recommended by a reviewer, (ii) take in input the submitted code and a reviewer comment posted in natural language and automatically implements the change required by the reviewer.

Despite the large amount of tasks on which DL models have been tested, to the best of our knowledge, there is still no approach aimed to improve code readability, probably because of the lack of a dataset. More generally, most studies of the application of DL on source code have not focused on improving its quality. We advance the state of the art by introducing a dataset of code readability improvement changes experimenting to what extent DL can be used for this task.

III. Extracting Readability Improving Commits

In this section, we present our methodology to build a dataset of readability-improving commits. To do this, we rely on the information provided by developers in commit messages. The first assumption on which our methodology relies is that commit messages reflect the developer's intention: If developers say they are improving code readability, we assume that their intention is to do so. The second assumption is that developers know how to improve code readability: If developer's intention is to improve code readability, the modification makes the code (even slightly) more readable. Both such assumptions were frequently used in previous work. For example, the SZZ algorithm [45] relies on information provided by developers to get the bug-inducing commits from a given issue. Our goal is not to select all the commits that improve code readability. Instead, since our final objective is to train a readability-improving model, we want that the commits we select actually improve code readability, to avoid misleading the model during training. In other words, we want to have high *precision*, but not necessarily high *recall*.

To build our dataset, we rely on GitHub Archive [13]. We extract a first set of candidate commits through keyword matching. Then, we apply several filters (including an NLP-based filter on commit messages) to discard false positives. A summary of the methodology we used is depicted in Fig. 1.

A. Step 1: Extracting Candidate Commits

As a first step, we want to select a set of commits that might aim at improving code readability. To do this, we mined GitHub Archive [13], a service which archives all the events occurring on GitHub, one of the most popular coding platforms. Specifically, we are interested in the *PushEvent*s, *i.e.,* the events occurring when a developer pushes their local commits to the remote repositories on GitHub.

Such events include, among other metadata, the list of pushed commits with their IDs and messages. At this stage, we only rely on the commit message to find candidate readability-improving commits by using a simple keyword-based approach. Several keywords might indicate that a commit aims at improving code readability (*e.g.,* "readability", "cleanup", "formatting"). However, after a preliminary analysis, we noticed that some of them result in very noisy samples, *i.e.,* they mostly select commits that do not (or do not only) improve code readability. For example, the word "formatting" is included in the commit message of `bc685` from the repository `needle-and-thread/vocal` [52], in which developers remove translation formatting. In this and other cases, the change is not related at all with code readability improvement. The best way we found to get a sample with a reduced amount of false positives was to include only commits explicitly referring to readability. Therefore, we only kept commits including "readab" (*i.e.,* "**readab**le", "**readab**ility", ...). This is in line with what was done in previous work [34]. We also noticed that, in some cases, the messages indicated that many different modifications were made, leading to tangled commits [16], [20]. This is particularly evident for "squash merges", *i.e.,* commits that summarize several commits on a given branch and that include the commit messages of all such commits by default. To reduce the risk of selecting tangled commits, we applied a filter to the commit message length. We kept only commits with less than 150 character.

B. Step 2: NLP-Based Filter

While keyword matching helps us selecting a good candidate set of commits, it also results in the inclusion of many false-positives. For example, in the repository `chelseacastelli/checklist`, commit `08cdb` [6], the developer improves the readability of the console output through the addition of coloring and text clearing. Such a modification changes a functionality instead of improving code readability. To discard false positives, we use an NLP-Based filter on the candidate commits selected after Step 1. Such a heuristic was inspired by the procedure used in previous work [42] and it was built through trial and error by manually testing it on a sample of commits filtered after Step 1.

The final heuristic we defined is based on a simple premise: We want to identify commit messages in which the term indicating readability is related to another term indicating improvement or to a term explicitly referring to a code element. We use the SpaCy [17] Python module for NLP to achieve this goal. First, we extract the sentences (S_m) from the commit message m. Also in this case, to reduce the risk of including tangled commits, we excluded commits with more than five sentences. Readability is often used to express properties of elements other than the source code. Therefore, we explicitly excluded commits related to GUI components (*e.g.,* readability of a web page) or to console printing (*e.g.,* readability of log messages)[1].

[1]The complete list of words we excluded is the following: "output", "notification", "doc", "user", "human"

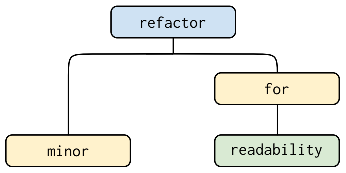

Fig. 2: Example of word dependency tree for the sentence "Minor refactor for readability"

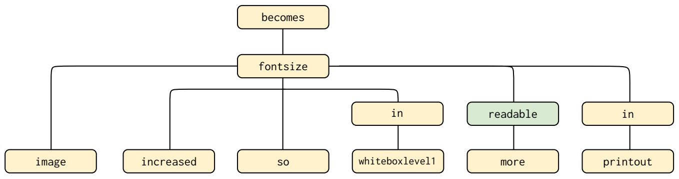

Fig. 3: Example of word dependency tree for the sentence "Increased fontsize in whitebox-level-1, so image becomes more readable in printout"

At this point, we defined three sets including the lemmas of the keywords of interest: K_r, including lemmas regarding the concept of readability, K_i, including lemmas regarding the concept of improvement, and K_c, including lemmas regarding code-related elements. K_r contains only "readab". K_i includes, instead, "improv" (*e.g.,* improved), "refact" (*e.g.,* refactoring), "enhanc" (*e.g.,* enhancement), "chang" (*e.g.,* change), "tweak", "keep", "increas" (*e.g.,* increase), "optim" (*e.g.,* optimized). Finally, K_c includes "code", "function", "variabl" (*e.g.,* variables), "statement", "line", "comment", "parameter", "instruct" (*e.g.,* instruction).

For each sentence $s_i \in S_m$ we used SpaCy to extract its word dependency tree t_i, which contains the syntactic relationships among the words composing the sentence. Based on the three sets K_r, K_i, and K_c, we mark all the words in t_i as *readability-related* (if matching any word in K_r), *improvement-related* (if matching any word in K_i), *code-related* (if matching any word in K_c), or *other*. We transform the dependency tree (t_i) in a directed acyclic graph (dag_i) in which an edge from node a to node b represents that b depends on a. If dag_i contains at least a path from any *readability-related* word to any *improvement-related* or *code-related* word, we keep the commit including such a sentence. If no sentence of a commit satisfies this condition, we discard it.

An example of word dependency tree for the sentence *"Minor refactor for readability"* is depicted in Fig. 2. In this example, the word "refactor" is marked as an *improvement-related* word, while the word "readability" is marked as a *readability-related* word. Since it exists a path that connects the two words (passing through the word "for"), we keep this commit. Another example is shown in Fig. 3 for the sentence *"Increased fontsize in whitebox-level-1, so image becomes more readable in printout"*. In this case, there is no path connecting the *readability-related* word ("readable") and the *improvement-related* word (*i.e.,* "increased"), which, indeed, refers to the font size rather than to code readability.

C. Step 3: File-Based Filter

The modifications made by developers to the repository files with a given commit constitutes an important piece of information that allows to further exclude commits that most likely did not aim at improving code readability. To acquire the list of modified files in each commit and the related patch, we use the official GitHub REST APIs [12]. Based on such a piece of information, we defined three strategies. In first strategy, we remove commits that change too many files (most likely tangled). To do this, we keep only commits that modify five or less files, as done in previous work [34]. In the second strategy, we discard commits that do not modify source-code files. We look at the names of the files modified in a given commit and we keep it only if it modifies at least a file containing source code. To detect if a file contains source code, we simply rely on its file extension. Specifically, we keep a commit if the extension of at least a modified file is included in a list of extensions related to the 10 most popular programming languages according to GitHut [3][2].

D. Step 4: Collecting Readability-Improving Modifications

For each commit c selected after Step 3, we download the version of each file modified in c both before and after the commit. We discard *added* and *removed* files (*i.e.,* we only keep *modified* files). Finally, we use the GitHub REST API [12] to get the main programming language of each repository included in our dataset and, later, we assign such a programming language to all the commits. We remove from our dataset all the commit that do not exist anymore in the repositories and, similarly, all the repositories that can not be found anymore (*i.e.,* deleted, renamed, or made private).

We executed the whole procedure on the history of GitHub between 2015-01-01 and 2022-12-31. In total, we acquired 122,045 readability-improving commits, totaling 156,348 file changes. Table I reports the number of commits and files for each year analyzed after each step of the our methodology. As it can be noticed, the number of readability-improving commits clearly grows in time, as it generally grows the number of commits on the whole GitHub platform. Also, the number of commits drastically reduces after each step. In total, $\sim$84% of the commits selected with the simple keyword matching (Step 1) gets discarded, as a result of the conservative approach we used.

We made our dataset publicly available [51]. It includes not only the final result (*i.e.,* the modified files), but also the relevant metadata (*e.g.,* name of the repository or commit author). We also release the scripts we used to build it so that researchers and practitioners can use them in the future to independently update the dataset (*e.g.,* to increase the training set for the model).

IV. Automatically Improving Code Readability

In this section, we describe our methodology we used to train a deep-learning model (T5 [40]) to automatically improve code readability.

[2]As for June 2021, when we started defining this heuristic.

TABLE I: Commits and files in time for each step.

Year	# Commits				# Files	
	Step 1	**Step 2**	**Step 3**	**Step 4**	**Step 3**	**Step 4**
2015	57,203	19,644	7,700	7,481	10,184	9,765
2016	67,007	23,225	9,597	9,301	12,723	12,185
2017	79,947	28,574	12,217	11,854	16,319	15,685
2018	86,869	31,883	13,789	13,458	18,509	17,837
2019	99,666	36,340	16,118	15,662	21,800	20,909
2020	121,019	45,621	20,689	20,253	27,926	26,959
2021	111,907	44,073	21,246	20,919	28,346	27,521
2022	126,711	50,124	24,243	23,117	30,684	25,487
Total	750,329	279,484	125,599	**122,045**	166,491	**156,348**

As a first step, given the modifications made by developers, we find matching pairs of code (*before* and *after* the modification), filter them, and tokenize the code so that it can be treated by the model. Then, we use such a dataset to tune, train, and test the model.

A. Datasets Definition

The dataset we built in Section III contains commits in which developers explicitly say that they improved code readability. However, such a dataset needs to be adapted to be used to train and test a machine learning model. We describe below the steps taken to achieve this goal.

Extracting Sub-Operations. A readability-improvement commit might change several files and, in each file, several lines of code. However, differently from bug fixing and refactoring operations, modifications aimed at improving code readability are intrinsically *local*. To fix a bug, a developer might change two classes, and such changes are dependent one from another. When improving readability, it is more likely that each change is independent from the other. For example, adding whitespaces before and after an operator improves code readability, if they are missing. While this operation could be repeated several times, it might be preferable to provide the model with examples easier to handle, from which it could more effectively learn how to achieve this goal. For example, let us consider commit `a7e6f` of the repository `perilstar/roomy` [36]. In this case, the same operation (the local variable `cg` was renamed in `channelGroup`) is performed in two different contexts. Our basic assumption is that *readability improvement is achieved by means of small, local improvements*. Therefore, the first step needed to define a dataset that a deep-learning approach can handle consists in the identification of such simpler operations. In this work, we extract them by using a simple heuristic. We first compute the diff between the version before and after the readability-improving commit. Then, we identify all the local modification sequences that involve up to 10 lines of code, and we take the two versions before and after the change. We discard all the changes occurred on files that do not match the target programming language. Since we are interested in building a model for Java, we only take into account files with extension ".java". The instances include not only the changed lines, but also 2 lines (at most) of context both before and after the modified lines.

Tokenization and Abstraction. To treat textual input/output in the T5 model, we need to tokenize it, *i.e.,* to split it in subsequences of characters. To achieve this goal, we use SentencePiece [21], an unsupervised technique that automatically detects recurring tokens. We also defined special tokens to abstract the input/output sequences (thus making the problem easier to solve). First, we replaced sequences of whitespaces before any instruction with a special `▷INDENTATION◁` token. Second, we replaced each remaining whitespace with a `▷WHITESPACE◁` token: We do this to force the model taking into account such elements, which in most of the other tasks (*e.g.,* bug fixing) are irrelevant. Third, we abstracted strings and numbers with `▷STRING◁` and `▷NUMBER◁` tokens, respectively: We do this because the specific strings/numbers might have a limited impact on code readability. We do not abstract other tokens (*e.g.,* identifiers and comments), instead, since they are important for readability [43].

Collecting and Splitting the Dataset. The dataset we built as described in Section III contains readability-improving commits in several programming languages. Different programming languages have different keywords, coding conventions, and syntax. Therefore, it is necessary to instantiate the model on each of them. In this work, we aim at building a model for Java. We choose such a programming language for several reasons. First, it is one of the most commonly used languages by developers and in software engineering studies [7], [14], [18], [22], [23], [29], [31], [48], [55]; second, it is one of the languages for which we have the largest amount of examples from our dataset (∼23k file modifications); third, the model we aim to use (T5) is pre-trained with Java code examples, and this allows us to partially re-use such a model. Given the readability-improving commits from the Java repositories, we run the previously described steps. Additionally, to have a high-quality dataset, we removed all instances that contain (i) `package` or `import` in the first 20 characters, to avoid making the model learn trivial operations regarding such elements; (ii) modification of TODO comments, which imply that new code was introduced; (iii) empty code either before or after the change, which indicate that new code was added or unneeded code was removed. Note that the abstraction step could result in making some instances useless: If, for example, the change regards a string, the *before* and *after* version of the instance are equal since both the concrete strings are replaced with a `▷STRING◁` token. We remove those instances. We also remove too trivial instances (*i.e.,* the ones with less than 10 tokens either before or after the change) and too long ones (*i.e.,* with more than 512 because we set this as maximum number of tokens for T5 as done in other work [28], [49]). Finally, we handled the cases in which different instances have either the same *before* or the *after* values. Having, for example, two instances for which the *before* is equal but the *after* is different might confuse the model since it can not understand which operation it has to perform. In those cases, we only keep one instance for each group, *i.e.,* the one for which the edit-distance between the *before* and *after* versions is lower.

After having acquired and abstracted pairs of sequences before and after the readability-improving modification, we end up with 31,183 instances. We randomly split such instances in three sets, following the procedures and the proportions used in previous work [29]. Most of the dataset (24,945 instances, *i.e.,* 80% of the total) is used as a *training set* (D_{ft}) to fine-tune the model. Half of the remaining instances (3,119, *i.e.,* 10% of the total) are used as the *validation set* (D_e) for hyper-parameter tuning, while the other half (again, 3,119 instances, *i.e.,* 10% of the total) are used as the test set (D_t) for our study.

B. Training Procedure of T5

Raffel *et al.* [40] introduced the T5 model in the domain of Natural Language Processing. This approach is based on two phases: *pre-training*, which allows building a language model useful to address different downstream tasks, and *fine-tuning*, which specializes the model to perform a concrete task. Mastropaolo *et al.* [28], [29] showed that T5 allows to achieve state-of-the-art results for four different coding tasks. For this reason, we decided to use such a model for our specific coding task (readability improvement). We report below the procedures we used for pre-training the model, tuning the hyper-parameters, and fine-tuning it for improving code readability.

Pre-training. Since Mastropaolo *et al.* [29] already pre-trained the model for coding-tasks, we did not repeat such a phase for our task. In the original work [29], the pre-training was done on the CodeSearchNet dataset, including both source code ($\sim$1.5M methods written in Java) and natural language sentences (499,618 sentences from the documentation of the methods). In total, the model was pre-trained on 2,984,627 instances.

Fine-tuning. We used the training set built as previously described to fine-tune the T5 model and, therefore, to obtain our readability-improvement model. We fine-tuned the model for 500k steps. In total, this phase took about $\sim$5 days of training on a Google Colab instance. We adopted the early stopping strategy [39] with the aim of avoiding overfitting. Specifically, every 10,000 fine-tuning steps, we evaluated the accuracy and took the best model before the accuracy decreased.

Decoding Strategy. The output layer of the T5 model needs to be decoded to provide the final output sequence(s). We use *beam-search* as a decoding strategy. When generating the output sequence, a *beam-search* strategy of order K keeps K hypotheses, *i.e.,* the K sequences of output tokens with the highest probability. Therefore, such a strategy will provide K token sequences. As also noted by Mastropaolo *et al.* [29], this is compatible with what developers might expect from a model that allows to address coding tasks. For example, the code completion feature provided by IDEs typically recommends many alternatives. This is particularly true for code readability, which is subjective by nature and for which no single solution might necessarily exist.

TABLE II: Types of learning rate used for hyper-parameter tuning, the respective parameter values, and the results.

Learning Rate Type	Parameters	Accuracy@1	BLEU-A
Constant	$LR = 0.001$	7.9%	0.53
Inverse Square Root (ISR)	$LR_{starting} = 0.01$ $Warmup = 10\,000$	13.8%	0.67
Slanted Triangular (ST)	$LR_{starting} = 0.001$ $LR_{max} = 0.01$ $Ratio = 32$ $Cut = 0.1$	**18.2%**	**0.75**
Polynomial Decay (PD)	$LR_{starting} = 0.01$ $LR_{end} = 0.001$ $Power = 0.5$	8.6%	0.54

Hyper-parameter tuning. To tune the hyper-parameters of the model, we tested it with four different types of learning rates, similarly to previous work [28]: (i) Constant Learning Rate, which consists in keeping the learning rate constant at a given LR value throughout the training; (ii) Inverse Square Root Learning Rate (ISR), which keeps the learning rate constant at $LR_{starting}$ for *Warmup* steps, and then it makes it decay as the inverse square root of the training step; (iii) Slanted Triangular Learning Rate (ST), which consists in starting from $LR_{starting}$, linearly increase it up to LR_{max}, after which it starts to linearly decrease; (iv) Polynomial Decay Learning Rate (PD), through which the learning decays polynomially from an initial value $LR_{starting}$ to a final value LR_{end}. We used the values reported in Table II.

We tuned the model for a total of 100k steps for each strategy, and we tested it on the validation set of our dataset. For each final model, we computed the effectiveness through two metrics: (i) Accuracy@1, which computes the percentage of *perfect predictions*, *i.e.,* cases in which a model using beam-search of order 1 as a decoding strategy is able to generate exactly the expected output; (ii) *BLEU-A score* [35], which measures the similarity between the expected output and the actual output in the $[0, 1]$ range. We provide more precise definition for such metrics in Section VI.

Table II reports the results. The Slanted Triangular Learning Rate (ST) allows to achieve the best results for both the metrics. This result agrees with previous results on different coding tasks [28], [29]. Thus, we decided to use this learning rate strategy in our model.

V. Study 1: Dataset Reliability

The *goal* of our first study is to validate the methodology we defined to automatically retrieve readability-improving commits and, therefore, the dataset we defined. This study is steered by the following research question:

RQ_1: *How precise is our approach in identifying readability-improving commits?* With our research question, we want to find out to what extent the defined dataset is reliable, *i.e.,* how many false-positives it wrongly selects.

A. Context Selection

To validate the dataset, we randomly selected a significant sample of modifications of any programming language.

The sample size we chose is 500 commits from our dataset, since this allows us to have a representative sample of our dataset: With a confidence level of 95%, the expected margin of error is ± 4.38. For each commit, we extracted (i) the link to the GitHub page showing the diff of the commit and (ii) the commit message.

B. Experimental Procedure

To answer RQ_1, three of the authors evaluated all the instances, aiming at assessing if the commit was about code readability improvement or not. To do this, we first equally divided the 500 selected commits in three bins, two of which with 167 instances and one with 166. Then, we assigned each evaluator with two bins, so that each evaluator had two different bins of instances to evaluate. This ensured that each instances was always reviewed by two evaluators. Each evaluator independently analyzed their own instances and reported, for each of them, "Yes" if the commit was improving readability and "No" otherwise. After the independent evaluation, we merged all the evaluations and we checked the disagreements. All the evaluators discussed such cases to reach a consensus.

To evaluate the instances (both in the first and second phase) we relied on the commit message and on the code diff. If the code modification did not only change the source code, but also the behavior of the code (*e.g.,* fixed a bug), regardless of the fact that this was mentioned in the commit message, we marked the commit as non-readability-improvement. We preliminarily set commits as "non-readability-improvement" when the message clearly indicated that the improvement was not related to code readability (*e.g.,* "improved web page readability"). We discarded commits that were not available anymore for at least one of the evaluators at the time of the evaluation. It is worth noting that our aim was not to check if the change actually improved readability, but rather if the *intention* of the developers was to do so. Therefore, we did not arbitrarily judge as "non-readability-improvement" the commits for which we personally believed that the change was not improving code readability.

We report the percentage of instances correctly improving code readability and we discuss cases in which our approach incorrectly included other commits.

C. Results

We discarded 7 instances that were not available on GitHub at the time of the evaluation for at least one of the evaluators. This lead to a total of 493 evaluated instances (986 evaluations, in total). The evaluators disagreed on 86 instances, which were carefully reviewed by the three evaluators in a meeting. In total, the evaluators agreed that 426 instances out of 493 were represented by readability-improvement commits (86.4%). Since the actual sample we selected is smaller, we re-computed a-posteriori the expected margin of error, which is slightly higher (± 4.4). Therefore, we can conclude that the percentage of actual readability-improving commits in our dataset is between 82% and 90.8% (95% confidence level).

D. Discussion

We discuss below the main causes for which our approach wrongly included commits that do not regard code readability.

Ambiguous commit message. Almost all the commit messages (included the false positives we identified) were believably related to the improvement of readability. We were not able to discard almost any commit based only on the commit message. Let us consider, for example, the repository `HelgeRottmann/r2dbe_software`, commit `dba8c` [15], with commit message "Various changes to improve the readability." Such a note might be reasonably associated to a commit which aims at improving code readability. Instead, the modifications suggest that the aim of the developers was to improve the readability of the program output. The fact that no commit message alone allowed us to directly mark a commit as a false-positive shows that the NLP heuristic we define works as intended. At the same time, other constraints might be added to make the heuristic even more conservative. For example, it might be worth experimenting the introduction of a constraint that forces the heuristic to select only commits in which *readability-related* terms are connected to both *improvement related* **and** *code-related* terms.

Misleading commit message. Sometimes, the commit message is not enough to understand if a commit improves code readability, and it might even mislead the NLP filter (or even a human annotator). Let us consider, for example, commit `9612b` from the `mat2m10/nerdsquad` repository [30]. The commit message reports "corrected some styling offenses & improved code readability." By reading the message alone, it is quite clear that the intention of the developers was to improve code readability. When looking at the modification, however, it can be noticed that the developer simply added two blank lines at the end of a file that contains many blank lines (85). It is quite clear that such a modification does not alter at all code readability. When looking at the history of such a repository, we found that most of the commit messages are the same, with analogous modifications (*e.g.,* added or removed blank lines at the end of the file). Indeed, our manually evaluated sample contains 8 of such commits. It might be worth adding a heuristics to discard trivial changes like this to improve the accuracy. It is worth noting that adding a blank line does not increase readability in this specific context (since it is added at the end of the file), but, in general, the number of blank lines might affect code readability [4], [5] when added to separate code blocks. In other words, simply discarding blank lines is not an option.

Tangled commits. While our approach uses several heuristics for reducing the amount of tangled commits, such commits still appear in the final dataset. This happens in commit `94b96` from the repository `Jaycean/apisix-dashboard` [19] with commit message "Fix bugs and change the readability of E2E test cases." While most of the modifications done by the developer improve code readability, some of them aim at fixing a bug, *i.e.,* they change the behavior of the program.

While we found that tangled commits are a minority in our dataset, they might negatively affect the model we aim to build. Therefore, further heuristics might be added to reduce the entity of such a problem. For example, commits with messages including keywords such as "bug" might be discarded.

VI. Study 2: Readability-Improving Model

The *goal* of our second study is to evaluate our readability-improvement model. This second study is steered by the following research questions:

RQ$_2$: *How does the proposed model perform compare to DL models designed to perform other coding tasks?* With this first research question, we aim at exploring how feasible is using DL (T5, in our case) to automatically improve code readability by comparing it with the same model used for other coding tasks.

RQ$_3$: *To what extent does the proposed model try to improve code readability?* With this second research question, we want to understand whether the model tries to improve readability or it makes other operations that, for example, change the behavior of the source code.

RQ$_4$: *To what extent does the proposed model improve code readability?* With this final research question, we aim at understanding whether the model actually improves code readability when it does not change the input code behavior.

A. Experimental Procedure

We use our readability-improvement model defined, trained, and tuned as reported in Section IV for all the RQs of this second study.

To answer *RQ*$_1$, *i.e.,* to provide a rough idea about the potentiality of our model, we compare its results with the ones achieved in previous work [28] on other code-related tasks. More specifically, we compare our model with the T5 models trained and tested by Mastropaolo *et al.* [28] for tasks that aim at generating source code. Therefore, we consider for comparison the following tasks:

- **Automated Bug Fixing (BF):** Given a buggy version of the source code as input, the model generates the correct version as output. There are two BF models, one for small methods (up to 50 tokens), and one for medium methods (up to 100 tokens).
- **Injection of Code Mutants (MG):** Given a correct version of the source as input, the model generates a buggy version as output.
- **Generation of Asserts in Tests (AG):** Given a partial test case as input, the model generates assert statements for the test. Results for T5 are available for two models: one that considers abstracted code, and one that considers raw code.

While Mastropaolo *et al.* [28] also consider another coding task (*i.e.,* Code Summarization), we do not include it in this comparison since it generates natural language text instead of code. We compute the following metrics for our model.

Accuracy@K. As previously reported, we use beam-search as a decoding strategy, with $K \in \{1, 5, 10, 25, 50\}$.

For a given K, given an input sequence s, a set of predictions provided by our model $ri^*_K(s)$, and the actual modification performed by developers, $ri(s)$, we say that a single prediction $ri^*_K(s)$ is correct if $ri(s) \in ri^*_K(s)$, *i.e.,* one of the predictions is exactly equal to the actual modification. Accuracy@K is computed as the percentage of correct predictions for the test set, for a given beam size K. We compare the results obtained by our model with the ones obtained for all the other tasks we consider (*i.e.,* BF, AG, and MG). As for the latter, we only compare the results of Accuracy@1, which is the only one available [28].

BLEU score. The BLEU-n score (Bilingual Evaluation Understudy) [35] aims at measuring how similar two texts (candidate and reference) are. Given a number n, n-grams are extracted from both the texts, and BLEU computes the percentage of n-grams from the candidate that appear in the reference. Such a metric ranges between 0 (candidate is unrelated with reference) and 1 (candidate is equal to reference). We compute BLEU-n, with $n \in \{1, 2, 3, 4\}$, and we report BLEU-A, *i.e.,* their geometric mean. We compare the values obtained by our readability-improving model with the ones obtained for the MG task, *i.e.,* the only one for which previous work reports such a metric.

To answer *RQ*$_2$ and *RQ*$_3$, we manually evaluated a significant sample of the predictions performed by our model composed of 500 instances in total (4.38 margin of error, 95% confidence level). To this end, we considered both perfect predictions (127 instances) and non-perfect predictions (373 instances), randomly sampled from the test set. Two of the authors independently analyzed each instance. They answered two questions: Q1, *i.e.,* "*does the predicted modification try to improve code readability?*" and Q2, *i.e.,* "*how does the change impact code readability?*" Q1 is a binary question ("yes" or "no"): Note that we use "no" to indicate both cases in which the model changes the behavior of the code (*e.g.,* it introduces new code) or it does not change the code at all. On the other hand, Q2 is a 5-point Likert scale question, for which the evaluators could assign a score from -2 (drastically decreases readability) to $+2$ (drastically increases readability). On this scale, 0 indicates that readability did not significantly change.

A third annotator manually evaluated the cases in which the two annotators disagreed. In Q1, the two annotators disagreed when they answered in an opposite way; in Q2, we consider that the annotators disagreed if one of them assigned a positive score and the other one assigned a negative score or if one of them assigned an positive/negative score and the other one assigned a 0. As for Q1, the third evaluator served as a tiebreaker: We determine the ground-truth "yes"/"no" answer by using majority voting. As for Q2, the third evaluator assigned, again, a subjective score. Regardless of the fact that there was agreement or not, we define the ground-truth answer to Q2 by computing the median among the three/two scores since we are handling an ordinal categorical variable. Note that while the first two evaluators answered both Q1 and Q2 for all the instances, the third evaluator did not answer Q2 when the majority of answers of Q1 was "no."

To answer RQ_3, we count the percentage of the answers to Q1, while to answer RQ_4 we count both the percentage of cases for which the final score was greater than 0 (*i.e.,* the model increased code readability), lower than 0 (the model decreased readability), and equal to 0 (*i.e.,* the model did not change code readability). Finally, we plot the distribution of the scores assigned using a barplot to show to what extent the model increased/decreased code readability.

B. Results

***RQ*$_2$: Model Effectiveness**. We report in Table III the results of our model (RI) and the state-of-the-art results obtained with the same T5 model for the other tasks we consider in our comparison. In terms of Accuracy@K, readability improvement ranges between 21% ($K = 1$) and 28% ($K = 50$). When comparing the results with the ones obtained with the same model used for other tasks, it can be noticed that with $K = 1$ (*i.e.,* single prediction), our readability improvement model performs similarly to the others (21% accuracy@1). When we observe the results obtained with higher values of K, the accuracy increases less than other models: RI flattens after $K = 5$. The difference in terms of accuracy obtained with $K = 5$ and $K = 50$ is very low (only three percentage point). We can conclude that generating more than 5 alternatives for code readability improvement unlikely would give developers practical advantages, as opposed to what happens for other tasks. In terms of BLEU-A, our model achieves a score of 0.77, which is, again, very similar to the score achieved for the mutant generation task (0.78).

In summary, our readability-improving model achieves acceptable results, comparable with the ones achieved by state-of-the-art models for performing other code-related tasks.

***RQ*$_3$: Tentative Readability Improvements**. Table IV reports the results of RQ_3. The model tries to improve the readability of the input code in ∼78% for perfect predictions. The result is much more negative (as it could be expected) for non-perfect predictions (only 19%). Overall, only 34% of the predictions actually try to improve code readability. To analyze more in depth this phenomenon, we specifically distinguished the cases in which the model changes the behavior (which might be dangerous for the developers who use the model) and the ones in which it does not change the code at all (which do not risk to harm the source code). We found that for 47% of the non-perfect predictions the model simply returns the input code although, as described in Section IV, no pairs with before and after equal were fed into the model. This means that, overall, our model changes the behavior of the code in only 31% of the cases. To put this result into perspective, we manually analyzed the 500 instances using the same methodology used to answer RQ_3 but aiming at checking in how many instances the original developers tried to improve readability (similarly to what we did in RQ_1). We found that in 28% of the instances the modifications made by the developers change the behavior, *i.e.,* they are not (only) aimed at improving readability. Thus, this behavior of the model is likely inherited from the noise in the dataset.

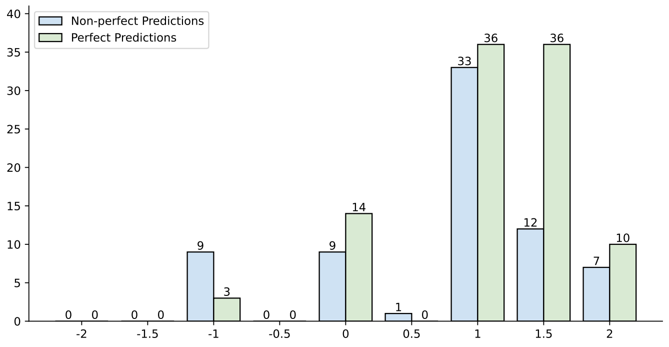

Fig. 4: Distribution of the readability scores assigned.

TABLE III: Results of RQ_2, where RI indicates our task. Missing data (–) indicate that the original work did not report such a piece of information.

Task	**Accuracy@K**					**BLEU-A**
	1	**5**	**10**	**25**	**50**	
RI	21%	25%	26%	27%	28%	0.77
BF (small)	10%	36%	44%	55%	60%	—
BF (medium)	3%	17%	24%	32%	38%	—
AG (abstracted)	34%	53%	58%	63%	66%	—
AG (raw)	47%	57%	61%	63%	65%	—
MG	28%	—	—	—	—	0.78

Finding ways to further remove such commits might result in better effectiveness of the model. Besides, to reduce the impact of this behavior and "force" the model to provide diversified predictions, it would be possible to use different values for the temperature hyper-parameter. Note that changing the temperature would have an impact mostly in the scenario in which a single prediction is considered. This is why we did not test different temperature levels in our experimental setting (we already use Beam Search with up to 50 predictions). Anyway, tuning such a parameter might be crucial in practice, when only a few suggestions can be presented to developers.

In summary, our readability-improving model should be used carefully, but it is generally quite reliable and it seldom changes the behavior of the input code.

***RQ*$_4$: Actual Readability Improvement**. Table IV reports the number of cases in which we observed an increase/decrease in code readability ($R^+_\sim$ and $R^-_\sim$, respectively) and the cases in which it does not significantly change ($R^0_\sim$) when the model tries to improve readability. It is quite clear that the model successfully achieves its goal: It increases readability, overall, in 79.4% of the cases, and it decreases it in a negligible percentage of cases (7.1%). These results are clearly higher for perfect predictions, but not significantly higher. This shows that even non-perfect predictions, when they do not change behavior, are valid. We report in Fig. 4 the distribution of the median scores assigned by the evaluators. Most of the scores are 1 and 1.5. This shows that some of the suggestions significantly improve code readability.

TABLE IV: Results of RQ_3 and RQ_4, where $R^{-}_{\sim}$, $R^{+}_{\sim}$, and $R^{0}_{\sim}$ indicate, respectively, the number of cases in which readability decreased, increased, and did not change.

	RQ_3	RQ_4		
		$R^{-}_{\sim}$	$R^{+}_{\sim}$	$R^{0}_{\sim}$
Perfect predictions	78.0%	3.0%	82.8%	14.1%
Non-perfect predictions	19.0%	12.7%	74.6%	12.7%
Overall	34.0%	7.1%	79.4%	13.5%

C. Discussion

Our results show that (i) the model achieves, in merely numerical terms, results comparable to the state-of-the-art models that address other coding tasks, (ii) it changes the behavior in a small percentage of cases, and (iii) when it does not change the behavior, it improves readability in the large majority of instances considered.

To understand how the model improves readability, we further analyzed the 500 instances considered to answer RQ_3 and RQ_4. We assigned one or more labels to each instance describing the types of operations performed and we later categorized them in three classes: *structure* (*e.g.,* a new variable has been introduced), *identifiers* (*e.g.,* renaming variables) and *comments* (*e.g.,* added comments). We found that most of the changes are aimed at modifying the structure (49%), some of them at improving identifiers (38%) and a few of them act on the comments (13%). This analysis further explains why the model sometimes changes the behavior of the input code: Changing the structure might result in non-equivalent transformations, especially for LLMs, which do not have any capability of formally checking the program equivalence. In addition, it is worth noting that changes related to the improvement of identifiers and comments have been studied individually [26], [27]. However, our approach incorporates both operations, plus other formatting-related ones, and aims at suggesting when each of them is needed, based on the context.

We show in Fig. 5 an interesting example. The model recognizes that one of the comments is not necessary in this context because it contains unused code; thus, it decides to remove it. It is worth noting that, in this specific case, the model performed better than the original developer, who tried to improve readability by further commenting out code (the `System.out.println(entries)` statement).

```
//Input sequence
try{
    for (String entries : fileContents){
        System.out.println(entries);
        //entries = entries.replace(""\n"", """"); Debug
     Temp disable
        String[] orders = entries.split($STRING$);
        if(!orders[$NUMBER$].startsWith($STRING$)){

//Prediction
try{
    for (String entries : fileContents){
        System.out.println(entries);
        String[] orders = entries.split($STRING$);
        if(!orders[$NUMBER$].startsWith($STRING$)){
```

Fig. 5: Example of successful readability improvement.

VII. Threats to Validity

We report below the threats to the validity of our studies.

Threats to Construct Validity. A possible threat to construct validity regards the fact that, in defining the dataset, we did not consider the quality of projects, as done in previous work [34]. This might result in the inclusion of low-quality samples (as discussed in Section V). We did this because our assumption is that readability can be improved also in small projects. Readability improvement operations might be more frequent at the beginning of a project, while are less frequent in mature project [37].

Besides, as described in Section V, our manual analysis of the dataset shows that most of the commits from our dataset actually improve readability. Our manual analysis showed the presence in our dataset of tangled commits, as discussed in Section V). This might constitute a threat to the validity for our second study, in which such commits might be included as instances in the training, validation, and test sets, resulting in lower performances of the model. In our sample, however, tangled commits constitute a minority (∼4.2%). We manually checked the 500 instances used to answer RQ_3 and RQ_4 to check the quality of the dataset built to train and test the model. We found that in 28% of them change the behavior, and they are more prevalent in non-perfect predictions (30%). Another possible threat to construct validity regards the definition of the dataset used to train, tune, and test the T5 model (Section IV). We included only a subset of the readability-improving operations extracted from the commits, *i.e.,* the ones have a limited impact on the code (at most ten lines). For example, complex refactoring operations might be ignored since they involve larger code blocks or files. We believe this is acceptable because specialized tools exist for supporting developers in refactoring operations [1], [24], some of them are also included in the IDEs. Finally, a possible limitation is related to the presence of nonexistent commits in the dataset. Since we defined the dataset, some commits have became not available (*e.g.,* they were deleted or the repository does not exist anymore). In the manually analyzed sample (RQ_1), this happened in 1.4% of the cases.

Threats to Internal Validity. A possible threat to internal validity can be the presence of human errors in the manual evaluation in both our studies. To reduce the impact of this problem, each instance was evaluated by at least two authors. In case of disagreement, all the authors reviewed again the instances to reach consensus (RQ_1) or a third author made an additional evaluation (RQ_3 and RQ_4).

Threats to External Validity. As for the first study, one of the threat to external validity is due to the choice of programming languages considered. Our dataset contains instances for 10 programming languages. In our second study, we tested our model for a single programming language, *i.e.,* Java. Using different programming languages might change the results.

VIII. Conclusion and Future Work

We defined (i) an automated approach for building a dataset of readability-improving commits and (ii) trained, for the first time, a LLM for automatically improving code readability. The results of our large empirical evaluation performed to validate both the approaches shows promising results. First, between 82% and 90.8% of the commits in the dataset actually aim at improving code readability. Second, our model is able to perfectly mimic how developers improve readability in 21% of the cases. Our model changes the behavior of the input code in only 31% of the cases, and, when it does not and it changes the code, it succeeds improving readability in 79% of the cases. We publicly release our dataset and all the scripts used to build it, together with the results of our manual investigations, to foster future research in this field [51].

In future work, we will first aim at improving the methodology we defined to build the dataset to remove as many false positives as possible by (i) further restricting the query and (ii) introducing other heuristics based on the results of our manual analysis. Furthermore, we plan to test additional LLMs, such as CodeT5 [53]. In the same spirit, we will compare our approach with ChatGPT: Such a tool can be asked to improve the readability of an code snippet, but its effectiveness in this task has never been assessed. Finally, we will extend our readability improvement approach to other programming languages (*e.g.,* Python, C, C++) to pave the way for the implementation of a comprehensive tool for readability improvement.

References

[1] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Learning natural coding conventions," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 281–293.

[2] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–37, 2018.

[3] F. Beuke, "Github Language Stats," https://madnight.github.io/githut/.

[4] R. P. Buse and W. R. Weimer, "A metric for software readability," in *Proceedings of the 2008 international symposium on Software testing and analysis*, 2008, pp. 121–130.

[5] ——, "Learning a metric for code readability," *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 546–558, 2009.

[6] chelseacastelli, "Checklist," https://bit.ly/3JtM5ww.

[7] Z. Chen, S. J. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, 2019.

[8] M. Ciniselli, N. Cooper, L. Pascarella, A. Mastropaolo, E. Aghajani, D. Poshyvanyk, M. Di Penta, and G. Bavota, "An empirical study on the usage of transformer models for code completion," *IEEE Transactions on Software Engineering*, vol. 48, no. 12, pp. 4818–4837, 2021.

[9] J. Dorn, "A general software readability model," *MCS Thesis available from (http://www. cs. virginia. edu/weimer/students/dorn-mcs-paper. pdf)*, vol. 5, pp. 11–14, 2012.

[10] L. Erlikh, "Leveraging legacy system dollars for e-business," *IT professional*, vol. 2, no. 3, pp. 17–23, 2000.

[11] S. Fakhoury, D. Roy, A. Hassan, and V. Arnaoudova, "Improving source code readability: theory and practice," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 2019, pp. 2–12.

[12] GitHub, "GitHub Get Commit API," https://docs.github.com/en/rest/reference/repos#commits, 2010.

[13] I. Grigorik, "GitHub Archive," https://www.gharchive.org/, 2012.

[14] S. Haque, A. LeClair, L. Wu, and C. McMillan, "Improved automatic summarization of subroutines via attention to file context," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 300–310.

[15] HelgeRottmann, "R2DBE Software," https://bit.ly/360QFnJ.

[16] K. Herzig and A. Zeller, "The impact of tangled code changes," in *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2013, pp. 121–130.

[17] M. Honnibal, I. Montani, S. Van Landeghem, and A. Boyd, "spaCy: Industrial-strength Natural Language Processing in Python," 2020.

[18] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 2018, pp. 200–20 010.

[19] Jaycean, "APISIX Dashboard," https://bit.ly/3t9jZRD.

[20] H. Kirinuki, Y. Higo, K. Hotta, and S. Kusumoto, "Hey! are you committing tangled changes?" in *Proceedings of the 22nd International Conference on Program Comprehension*, 2014, pp. 262–265.

[21] 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.

[22] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *Ieee transactions on software engineering*, vol. 38, no. 1, pp. 54–72, 2011.

[23] A. LeClair, S. Jiang, and C. McMillan, "A neural model for generating natural language summaries of program subroutines," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 795–806.

[24] B. Lin, S. Scalabrino, A. Mocci, R. Oliveto, G. Bavota, and M. Lanza, "Investigating the use of code analysis and nlp to promote a consistent usage of identifiers," in *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2017, pp. 81–90.

[25] B. Loriot, F. Madeiral, and M. Monperrus, "Styler: learning formatting conventions to repair checkstyle violations," *Empirical Software Engineering*, vol. 27, no. 6, p. 149, 2022.

[26] A. Mastropaolo, E. Aghajani, L. Pascarella, and G. Bavota, "An empirical study on code comment completion," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021, pp. 159–170.

[27] ——, "Automated variable renaming: are we there yet?" *Empirical Software Engineering*, vol. 28, no. 2, p. 45, 2023.

[28] A. Mastropaolo, N. Cooper, D. N. Palacio, S. Scalabrino, D. Poshyvanyk, R. Oliveto, and G. Bavota, "Using transfer learning for code-related tasks," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 1580–1598, 2022.

[29] A. Mastropaolo, S. Scalabrino, N. Cooper, D. N. Palacio, D. Poshyvanyk, 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.

[30] mat2m10, "nerdsquad," https://bit.ly/37xI2Bw.

[31] A. Mesbah, A. Rice, E. Johnston, N. Glorioso, and E. Aftandilian, "Deepdelta: learning to repair compilation errors," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 925–936.

[32] Q. Mi, J. Keung, Y. Xiao, S. Mensah, and Y. Gao, "Improving code readability classification using convolutional neural networks," *Information and Software Technology*, vol. 104, pp. 60–71, 2018.

[33] R. Minelli, A. Mocci, and M. Lanza, "I know what you did last summer-an investigation of how developers spend their time," in *2015 IEEE 23rd International Conference on Program Comprehension*. IEEE, 2015, pp. 25–35.

[34] J. Pantiuchina, M. Lanza, and G. Bavota, "Improving code: The (mis) perception of quality metrics," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 80–91.

[35] 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.

[36] perilstar, "Roomy," https://bit.ly/3q5l7Uz.

[37] V. Piantadosi, F. Fierro, S. Scalabrino, A. Serebrenik, and R. Oliveto, "How does code readability change during software evolution?" *Empirical Software Engineering*, vol. 25, no. 6, pp. 5374–5412, 2020.

[38] D. Posnett, A. Hindle, and P. Devanbu, “A simpler model of software readability,” in *Proceedings of the 8th working conference on mining software repositories*, 2011, pp. 73–82.

[39] L. Prechelt, “Early stopping-but when?” in *Neural Networks: Tricks of the trade*. Springer, 2002, pp. 55–69.

[40] 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.

[41] RoboJackets, “Readability commit example,” https://github.com/RoboJackets/roboracing-software/commit/04207855914c7b55e2d47fbc2efd4ca293acfc16, 2019.

[42] G. Rosa, L. Pascarella, S. Scalabrino, R. Tufano, G. Bavota, M. Lanza, and R. Oliveto, “Evaluating szz implementations through a developer-informed oracle,” in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 436–447.

[43] S. Scalabrino, M. Linares-Vásquez, R. Oliveto, and D. Poshyvanyk, “A comprehensive model for code readability,” *Journal of Software: Evolution and Process*, vol. 30, no. 6, p. e1958, 2018.

[44] S. Scalabrino, M. Linares-Vasquez, D. Poshyvanyk, and R. Oliveto, “Improving code readability models with textual features,” in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. IEEE, 2016, pp. 1–10.

[45] J. Śliwerski, T. Zimmermann, and A. Zeller, “When do changes induce fixes?” *ACM sigsoft software engineering notes*, vol. 30, no. 4, pp. 1–5, 2005.

[46] A. Thies and C. Roth, “Recommending rename refactorings,” in *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, 2010, pp. 1–5.

[47] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, “Learning how to mutate source code from bug-fixes,” in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019, pp. 301–312.

[48] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, “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.

[49] R. Tufano, S. Masiero, A. Mastropaolo, L. Pascarella, D. Poshyvanyk, and G. Bavota, “Using pre-trained models to boost code review automation,” in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2291–2302.

[50] 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.

[51] A. Vitale, V. Piantadosi, S. Scalabrino, and R. Oliveto, “Replication package of ”using deep learning to automatically improve code readability,” 8 2023. [Online]. Available: https://doi.org/10.6084/m9.figshare.22774784.v1

[52] VocalPodcastProject, “Vocal,” https://bit.ly/3IfvMlU.

[53] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, “Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation,” in *EMNLP*, 2021.

[54] C. Watson, N. Cooper, D. N. Palacio, K. Moran, and D. Poshyvanyk, “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.

[55] C. Watson, M. Tufano, K. Moran, G. Bavota, and D. Poshyvanyk, “On learning meaningful assert statements for unit test cases,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1398–1409.

[56] Y. Yang, X. Xia, D. Lo, and J. Grundy, “A survey on deep learning for software engineering,” *ACM Computing Surveys (CSUR)*, vol. 54, no. 10s, pp. 1–73, 2022.