POLITECNICO DI TORINO
Repository ISTITUZIONALE

On the Impact of Code Comments for Automated Bug-Fixing: An Empirical Study

Original
On the Impact of Code Comments for Automated Bug-Fixing: An Empirical Study / Vitale, A., Guglielmi, E., Scalabrino, S., Oliveto, R.. - (2026), pp. 1-13. (ICPC '26: 34th IEEE/ACM International Conference on Program Comprehension Rio De Janeiro (BRA) April 12 - 13, 2026) [10.1145/3794763.3794794].

Availability:
This version is available at: 11583/3015067 since: 2026-09-01T16:05:00Z

Publisher:
Association for Computing Machinery

Published
DOI:10.1145/3794763.3794794

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

28 November 2026

On the Impact of Code Comments for Automated Bug-Fixing: An Empirical Study

Antonio Vitale
Politecnico di Torino
Torino, Italy
University of Molise
Termoli, Italy
antonio.vitale@polito.it

Emanuela Guglielmi
University of Molise
Termoli, Italy
emanuela.guglielmi@unimol.it

Simone Scalabrino
University of Molise
Termoli, Italy
simone.scalabrino@unimol.it

Rocco Oliveto
University of Molise
Pesche, Italy
rocco.oliveto@unimol.it

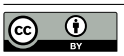
Abstract

Large Language Models (LLMs) are increasingly relevant in Software Engineering research and practice, with Automated Bug Fixing (ABF) being one of their key applications. ABF involves transforming a *buggy* method into its *fixed* equivalent. A common preprocessing step in ABF involves removing comments from code prior to training. However, we hypothesize that comments may play a critical role in fixing certain types of bugs by providing valuable design and implementation insights. In this study, we investigate how the presence or absence of comments, both during training and at inference time, impacts the bug-fixing capabilities of LLMs. We conduct an empirical evaluation comparing two model families, each evaluated under all combinations of training and inference conditions (*with* and *without* comments), and thereby revisiting the common practice of removing comments during training. To address the limited availability of comments in state-of-the-art datasets, we use an LLM to automatically generate comments for methods lacking them. Our findings show that comments improve ABF accuracy by up to threefold when present in both phases, while training with comments does not degrade performance when instances lack them. Additionally, an interpretability analysis identifies that comments detailing method implementation are particularly effective in aiding LLMs to fix bugs accurately.

CCS Concepts

• **Software and its engineering → Software development methods**; **Empirical software validation**; • **Computing methodologies → Machine learning**.

ACM Reference Format:
Antonio Vitale, Emanuela Guglielmi, Simone Scalabrino, and Rocco Oliveto. 2026. On the Impact of Code Comments for Automated Bug-Fixing: An Empirical Study. In *34th IEEE/ACM International Conference on Program Comprehension (ICPC '26), April 12–13, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3794763.3794794



This work is licensed under a Creative Commons Attribution 4.0 International License.
ICPC '26, Rio de Janeiro, Brazil
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2482-4/26/04
https://doi.org/10.1145/3794763.3794794

1 Introduction

Large Language Models (LLMs), are increasingly becoming indispensable tools for assisting software developers in their daily tasks. These models enable the automation of a wide range of activities. Furthermore, developers can easily leverage advanced LLM-powered tools, such as GitHub Copilot, which are seamlessly integrated into modern Integrated Development Environments (IDEs) to enhance productivity and streamline workflows.

Among these applications, Automated Bug Fixing (ABF) has emerged as one of the most researched and impactful coding tasks [5, 10, 26, 34, 39, 56, 68]. ABF involves identifying and resolving bugs within a method (*e.g.,* a scenario where a test case fails). While the concept may seem straightforward, ABF holds significant value for developers, as bug fixing is notoriously challenging and often demands considerable effort [15, 46]. By alleviating this burden, ABF has the potential to substantially enhance developer efficiency and reduce the time spent on error resolution.

Early research on using deep learning (DL) and LLMs for ABF established the foundation for the field. Tufano *et al.* [56] presented one of the first DL-based approaches to automatically fix bugs, quickly followed by several others [10, 34, 68]. A key contribution of the work by Tufano *et al.* [56] was the release of a benchmark dataset, which remains a standard for training and evaluating ABF models [3, 8, 14, 19, 23, 24, 39, 63, 65]. Such a dataset, mined from GitHub, contains couples of buggy Java methods and their respective fixed versions. Given the limitations of the specific ML model they adopted (Recurrent Neural Networks — RNNs), they needed to pre-process the source code of the methods so that identifiers are abstracted (*i.e.,* they appear like `IDi`, where `i` is an incremental number) and comments are removed.

However, code comments may not only be important [61], but often essential for fixing bugs. Comments, indeed, might contain precious information about the requirements, which are sometimes necessary to understand what is wrong with the code and, thus to fix it. Let us consider the example in Fig. 1. The buggy code lacks `mNearbyDeviceAdapter.notifyDataSetChanged()`. Such an instruction notifies the dataset change to the nearby devices. Without reading the comment and understanding this requirement, neither a human developer nor a model could identify or fix the bug.

```
@@ -1,11 +1,12 @@
/*
 * Empties the PWO metadata queue and notifies the dataset
 * change to the nearby devices.
 */
private void emptyPwoMetadataQueue() {
    Collections.sort(mPwoMetadataQueue);
    for (PwoMetadata pwoMetadata : mPwoMetadataQueue) {
        mNearbyDeviceAdapter.addItem(pwoMetadata);
    }
    mPwoMetadataQueue.clear();
+   mNearbyDeviceAdapter.notifyDataSetChanged();
}
```

Figure 1: Example of a buggy method and its fixed version (the green line is the fix, which was missing in the buggy version).

Unlike the RNNs employed by Tufano *et al.* [56], modern LLMs are well-equipped to process code comments. Despite this capability, prior research has overlooked the potential value of incorporating comments—a potentially critical source of information—for Automated Bug Fixing (ABF).

In this work, we investigate how the presence or absence of code comments, both during training and at inference time, impacts on the bug-fixing capabilities of Large Language Models (LLMs). First, we build a new dataset for ABF based on the one introduced by Tufano *et al.* [56]. To do this, we consider the same repositories and commits, but we re-extract the instances by considering the raw source code (*i.e.,* we do not abstract identifiers and keep the original comments). Differently from the previous work, which only considered *small*- ($\leq$ 50 tokens) and *medium*-sized ($\leq$ 100 tokens) methods in two distinct datasets, we merge them in a single dataset, in which we also include *big* methods ($\leq$ 512 tokens).

We noticed, however, that only a minority of the code instances in our dataset contained comments (< 20%) and that many of them are low quality (*e.g.,* "`// Argh!!! This method is WRONG !`"). For this reason, we use an automated procedure to generate higher-quality comments needed in our experiment. Considering the taxonomy of code comments introduced by Zhai *et al.* [69], we use an LLM to generate five comments, documenting **what** the method does, **how** it does it, the intended **usage** of the method, the reason **why** the method is there, and **properties** of the method (*e.g.,* pre-conditions and post-conditions). We do this on the fixed version rather than on the buggy one to simulate the ideal scenario in which developers properly document the methods they develop.

To evaluate our hypothesis that code comments play a crucial role in Automated Bug Fixing (ABF), we designed a controlled experiment manipulating the presence of comments both during training and inference. We fine-tuned two types of models—CodeT5+ [64] and DeepSeek-Coder [20]—under all combinations of these conditions (*with* vs. *without* comments). For the training phase, comments were either removed from or automatically generated for each instance, while at inference time the models were tested on inputs with or without comments accordingly.

When comparing the four training-inference combinations, we found that models benefit from comments in both phases. Comments provided only at inference time already lead to noticeable improvements, while training with comments does not harm performance on instances lacking comments. The greatest improvements occur when comments are present in both phases, increasing accuracy by up to threefold. These results demonstrate that code comments can substantially enhance models performance, underscoring their importance in ABF tasks.

To gain deeper insights into the impact of the five categories of comments on model performance, we conducted an interpretability study based on SHAP [33], employing the GradientExplainer method. This method assigns an importance score to each input token with respect to every token generated during the inference of a single generation. Our analysis revealed that the most influential category of comments were those documenting **how** the method achieves its goal.

Interestingly, the importance of other comment categories varied between the two models. For one model, *i.e.,* CodeT5+, comments describing **what** the method does were the least important, likely because they provide high-level information that is often redundant (*e.g.,* already conveyed by the method name). For the other model, *i.e.,* DeepSeek-Coder, the lowest importance was attributed to comments describing the method's **usage** and its **properties**.

In summary, our findings reinforce a recurring theme in Software Engineering research: *Developers should document their code not only for themselves and future collaborators but also to enable LLMs to provide better assistance.*

All code and data used in our study is publicly available [1].

2 Background and Related Work

We first present some background on Machine Learning (ML) for coding tasks, with specific focus on the Automated Bug Fixing (ABF) task. Then, we present some related works.

2.1 ML for Coding Tasks

Software Engineering ML tasks are typical software engineering activities that can be supported by Machine Learning. Among these, Coding ML Tasks (or simply Coding Tasks) concern source code. They may involve code classification (*e.g.,* vulnerability detection), regression (*e.g.,* bug-fix time prediction), or code generation (*e.g.,* automated bug fixing). We focus on the latter, which includes code summarization [2, 21, 29, 36], code completion [11, 54], and automated bug fixing [10, 56]. While several approaches tried to represent source code as trees (*e.g.,* AST) or graphs (*e.g.,* CFG) to devise ML-approaches able to perform specific tasks, most models simply represent source code as text. Thus, neural networks used in Natural Language Processing (NLP) tasks found their way into the SE community. The first approaches defined to tackle coding tasks were based on Recurrent Neural Networks (RNNs). RNNs are designed to process sequential data through recurrent connections that allow information to persist across sequence steps. Their basic input unit is the *token*, which represents a word, subword, or character, depending on the tokenizer used to convert text into token sequences. Since RNNs handle numeric vectors, tokens are transformed into vector representations through an *embedding* step.

The Transformer architecture [60] constituted a revolution in this field. Such an architecture has been adopted both for NLP [50] and coding tasks [37, 39, 65]. The Transformer architecture relies on a self-attention mechanism to capture long-range dependencies among tokens. Unlike RNNs, which process tokens sequentially, this architecture can handle longer context windows (*i.e.,* larger amounts of tokens) efficiently given its parallel tokens processing.

While the definition of *Large Language Model* (LLMs) is fuzzy, most use such a term to refer to any artificial neural network that relies on the Transformer architecture [60] to perform a specific task. Thus, we use such a term with this meaning from now on.

LLMs work in two steps: *pre-training* and *fine-tuning*. In the *pre-training* step, the model is trained to perform a self-supervised task (*e.g.,* predicting the next token in a sequence). This allows the model to learn the language(s) that it must treat. In the *fine-tuning* step, instead, the model is trained to perform a specific task (*e.g.,* to transform a given input sequence to a given output sequence). For example, for the ABF task on which we focus in this paper, the input sequence represent a *buggy* method, while the output sequence is the *fixed* version of the same method. Since 2021, several LLMs specialized in source code have emerged [37, 39, 64, 65]. Among them, CodeT5+ [64] extends the T5 architecture [50]. It is pre-trained on nine programming languages using multiple code-oriented objectives, including *span denoising*, and *causal LM*, and is available in sizes from 220M to 16B parameters. DeepSeek-Coder [20], instead, is trained from scratch on 2T tokens across 87 languages, building on the Llama architecture [55]. It employs *next-token prediction* and *fill-in-the-middle* pre-training objectives, and is available in base and instruction-tuned variants ranging from 1.3B to 33B parameters.

2.2 Automated Bug Fixing

One of the first studies regarding the use of DL to fix bugs is the one by Tufano *et al.* [56]. The authors use a Neural-Machine-Translation (NMT) approach to address ABF. They found that it was able to fix between 9% and 50% of the bugs from their dataset.

Lutellier *et al.* [34] proposed CoCoNuT, a program repair technique combining ensemble learning and context-aware NMT. The method uses convolutional neural networks (CNNs) to encode buggy code and context independently, enhancing accuracy and capturing a wide range of strategies. CoCoNuT was observed to outperform 27 strategies in four languages, fixing 509 bugs (309 of which were unique).

Previous work mostly rely on word-level tokenizers that split the input/output text into tokens based on words defined from a given vocabulary. Therefore, they inherently suffer from the out-of-vocabulary (OOV) problem: If a word not belonging to the vocabulary appears in the test data, it can not be handled and it is often replaced with a special token (*e.g.,* <UNK>). Also, they mostly relied on Recurrent Neural Networks, which are not capable of considering a large context while predicting tokens.

The introduction of Transformers and advanced tokenizers like Byte Pair Encoding (BPE) [28] allowed to overcome such limitations. Mastropaolo *et al.* [38, 39] relied on the T5 architecture to tackle several coding tasks, including ABF. Their results show that such an approach achieves better results than the approach by Tufano *et al.* [56] on their two datasets.

Jiang *et al.* [25] studied several *Code Language Models* – LLMs for coding tasks – *i.e.,* PLBART [3], CodeT5 [65], CodeGen [45], and InCoder [17] in the bug-fixing context. The authors showed that the best CLM, without fine-tuning, fixes 72% more bugs than the state-of-the-art deep learning-based techniques, while fine-tuning enables CLMs to fix 46%–164% more bugs than previous techniques.

2.3 Data Quality in Coding Tasks

A few studies analyzed the impact of training data quality on the resulting model. Liu *et al.* [31] addressed the method-name prediction task enriching the instances with (i) documentation, (ii) local context, and (iii) project-specific context. This allowed them to train a model which outperformed state-of-the-art approaches by a large margin. Prenner *et al.* [49] studied the importance of local context in instances for automated program repair models. Given different measures of context (*e.g.,* lines before the bug), the authors trained different Transformer models, finding that the repair success increases when the local context is bigger. The work most closely-related to ours is the one by Van Dam *et al.* [59]. The authors show that enriching the code instances for the code completion task with their original comments results in higher effectiveness. Differently from Van Dam *et al.*, we (i) focus on the ABF task, and (ii) do not rely on the original comments by the authors but on possibly higher-quality and more complete comments, automatically generated with GPT-3.5.

3 Theory: Why and What Comments Matter

In this section, we present our theory on the significance of comments for ABF through LLMs and outline the types of informative content in the comments that we expect could play a crucial role.

3.1 Why Comments Matter for ABF with LLMs

LLMs are, at their core, language models: Their goal is to predict the most likely token t_{n+1} given a *context* χ, which is a sequence of tokens $\langle t_1, \ldots, t_n \rangle$. The fine-tuning step, which teaches the model how to perform a specific task, is performed through a dataset Φ, in which each instance is represented as a pair $\langle input, output \rangle$, where both elements are token sequences. Let us consider the ABF task, *i.e.,* the task of transforming a given *buggy* method m_b into a non-buggy (or fixed) method m_f. In the fine-tuning step, the model is trained to generate, one at a time, the tokens in m_f (*output*) given the ones in m_b (*input*). LLMs are auto-regressive models [60]: For each token to generate, the context is constituted by the union of the *input* sequence (m_b for ABF) and the previously generated tokens, until a special token is generated to indicate the end of the *output* sequence. Formally, the context that is adopted to generate the token t_k can be recursively expressed as $\chi_{t_k} = \chi_{t_{k-1}} \cup G(\chi_{t_{k-1}})$, where χ_{t_0} is simply the input sequence (buggy method m_f in ABF) and G indicate the inference procedure that predicts the next token given a context.

Consider two identical methods, m_a and m_b, both consisting of a single line of Python code: `return sorted(list)`. The first method, m_a, is intended to sort a list of strings in a case-sensitive manner and is therefore bug-free. In contrast, m_b is meant to sort a

Table 1: Multi-intent comment taxonomy [9].

Category	Description	Example
What	Describes the functionality of a method	*"A helper function that processes the stack."*
Why	Explains the reason why a method is provided or the design rationale of the method	*"Get a copy of the map (for diagnostics)"*
Usage	Describes the usage or the expected set-up of using a method	*"Should be called before the object is used"*
How	Describes the implementation details of a method	*"Convert the byte[] to a secret key"*
Property	Asserts properties of a method including pre-conditions or post-conditions of a method	*"Wait until seqno is greater than or equal to the desired value or we exceed the timeout."*
Others	Unspecified or ambiguous comments	*"The implementation is awesome."*

list of strings in a case-insensitive manner, making it buggy. This example illustrates how the same source code can be simultaneously *bug-free* in one context and *buggy* in another. Now, let us introduce two models: M_{Φ_b} and M_{Φ_c}. The model M_{Φ_b} is fine-tuned on a dataset Φ_b that contains only code without comments, while M_{Φ_c} is fine-tuned on Φ_c, a dataset enriched with comments that describe the code in each instance. The difference between these models lies in their learning capabilities. M_{Φ_b} learns to generate code-related tokens solely based on other code-related tokens. In essence, it focuses on repeating patterns of changes within the code itself. Conversely, M_{Φ_c} is exposed to a broader set of patterns, enabling it to learn not only code-to-code transformations but also natural language-to-code mappings, akin to what happens in the code generation task. This additional context allows M_{Φ_c} to better infer the intended behavior of the code, particularly in cases where subtle requirements—such as sorting criteria—are conveyed through comments.

3.2 What Comments Matter

Code comments can be aimed at documenting different aspects, such as design rationale, pre-conditions/post-conditions, and implementation details [69]. According to a study by Chen *et al.* [9] comments can have six different intents, and they can be used to document **what** the method does, **why** it does it (*i.e.,* the design rationale), **how** the method is implemented, what is its intended **usage**, what **properties** characterize it (*e.g.,* pre- or post-conditions), or **other** aspects. We report in Table 1 a complete description of each category with examples. Mu *et al.* [44] recently observed that over 66% of comments in top-starred Java repositories are multi-intent, *i.e.,* they document multiple aspects of the code within a single comment. This suggests that developers often feel the need to capture various dimensions of a method behavior or purpose in their comments. We conjecture that comments addressing all the above-mentioned intents, except for *other*, can enhance the effectiveness of ABF models. Comments categorized as *other* primarily represent noise and do not contribute meaningful information about the code (see the example in Table 1). Therefore, we assume that comments with this intent play a minimal role in aiding bug fixing.

To better explain why we theorize this, let us consider the following example. There is a method named `processCSV` that reads a CSV file and processes each row to compute the mean of the numerical value contained in the i-th column, where i is a parameter. The method is supposed to count empty cells as 0, but it throws an exception instead since it reads an empty string which is not a number. Without comments, the model might attempt to fix the method by adding a check to skip empty strings entirely. However, this would not resolve the issue but merely change the nature of the bug, as the intended behavior of counting empty cells as 0 would still not be achieved.

A *property*-related comment could contain information about the post-condition (*e.g.,* "the method returns a number") that could help the model understand where the issue is (*i.e.,* it throws an exception). Comments documenting *what* the method is supposed to do (*e.g.,* "computes the mean of the values in the i-th column and counts the empty cells as 0") and *how* the method is implemented (*e.g.,* "the method sums the values in the i-th column over the rows and returns such a value divided by the number of rows") may help to clarify the intended functionality and find discrepancies between the code and the expected behavior. A comment documenting *why* the method is necessary (*e.g.,* "computes a robust mean of the values (for statistical calculations)") might help to indicate the model to handle edge-cases without missing values. Finally, a comment documenting the expected *usage* (*e.g.,* "the i value should refer to a column that only contain numerical data or empty values") could help guiding the model to avoid performing unnecessary checks (*e.g.,* if the cell contains non-numeric characters).

4 Empirical Study Design

The *goal* of our study is to validate our theory, *i.e.,* to determine whether the presence of comments can improve the effectiveness of LLMs in the ABF task. Thus, our study is steered by the following Research Questions (RQs):

- RQ_1: *Does adding comment allow to improve the bug fixing capabilities of LLMs?*
- RQ_2: *How important are the comments to the model?*

4.1 Study Context

The context of the study consists of buggy Java methods paired with their fixed versions. We mainly rely on the data released by Tufano *et al.* [56]. In their work, the authors mined GitHub from March 2011 to October 2017 and collected bug-fixing commits through matching patterns (*e.g.,* "fix bug", "solve error", etc.). In total, they collected ~10M of such commits. They filtered out commits that (i) contained non-Java files, (ii) created new files, and (iii) involved changes in more than five files. Thus, they reduced the number of commits to 787,178. They associated each *fixed* version of the method to its previous version (*buggy*). After removing comments and annotations, they further abstracted the pairs by replacing identifiers and literals with placeholders, preserving frequent idioms (*i.e.,* frequent identifiers and literals). Finally, they filtered the $\langle bug, fix \rangle$ pairs into two datasets based on the number of tokens, creating BFP_{small} (58k instances with at most 50 tokens) and BFP_{medium} (65k instances with at least 50 tokens, and at most 100 tokens). We selected this dataset because, among those publicly

available, it represents the most suitable and comprehensive option for our study. First, it provides bug-fix pairs at the *method-to-method* level, offering a more realistic granularity of software changes than *single-hunk* or snippet-based datasets commonly used in previous work (*e.g.,* the dataset by Zhu *et al.* [70]). Second, such instances cover a more recent and representative period (*i.e.,* 2011–2017), unlike older datasets such as those by Lutellier *et al.* [34] (pre-2010) or Monperrus *et al.* [43] (circa 2015). Finally, although the original dataset was released as two separate *small* and *medium* subsets, as we explain below, we merged and extended them into a single renovated version to overcome their original limitations.

Dataset Renovation. Despite the before-described datasets have been extensively leveraged as benchmarks in several studies that tackled ABF [3, 8, 14, 19, 23, 24, 39, 63, 65], they suffer from some limitations. Because of the limited capabilities of the models they used (*i.e.,* RNN-based) and because of the Out-of-Vocabulary (OOV) problem, the authors abstracted the identifiers of the instances and partitioned the datasets based on token count.

To overcome such limitations, we re-build the dataset from the raw source code. Starting with the ~787k mined commits provided in the replication package by Tufano *et al.* [56], we first extract all methods that underwent fix-related changes. Following the original methodology, we use the GumTree Spoon AST Diff tool [16] to calculate the number of edit actions (in terms of AST transformations). We exclude methods with no edit actions — *e.g.,* those involving only changes to comments — and those requiring more than one hundred edit actions, as these are likely tangled commits involving changes beyond a simple bug fix. This step filters out 176,413 instances. Additionally, we identify and remove 87,819 duplicate instances. We keep only instances from commits containing changes focused on a single method. We do this to avoid cases in which the fix requires edit action on more than a method. The model, indeed, will see only a single method and will be asked to fix it. This means that we exclude those bug-fixes for which the change in a method does not completely fix the bug, ensuring a fair training and evaluation process. We exclude ~1.5M instances in this step. Then, following the state-of-the-art pre-processing procedures, we perform additional filtering steps. Specifically, we exclude instances with commented-out code [52] (106,487), unusual sequences like "____" [58] (13,245), and self-admitted technical debt, like `TODO` or `FIXME` [35, 52, 58] (20,123). To avoid to confuse the model during fine-tuning, we remove instances for which we have the same "buggy" code and different "fixed" code, and vice-versa [62] (12,971).

Unlike Tufano *et al.* [56], we do not abstract identifiers or literals, preserving the source code in its original form. We exclude instances with fewer than 10 tokens (trivial methods) and those exceeding 512 tokens, as the model cannot handle more than this number of tokens. This filtering approach aligns with similar works [38, 57, 62]. As a result, our dataset includes larger instances compared to the dataset by Tufano *et al.* [56], which is limited to methods with a maximum of 100 tokens, as shown in Table 2. After applying these filters, we obtain a dataset comprising 116,372 bug-fix pairs, which we refer to as D_b.

Automatically Generating Comments. To validate our theory, we require two datasets: one containing commented instances and another with the same instances stripped of comments (see the experimental procedure for more details). A potential approach could have been to use only the instances in D_b that include developer-written comments as the dataset with comments, and create the dataset without comments by removing those comments. However, we observed that only a small fraction of instances in D_b contain developer-written comments (approximately 20%). Moreover, many of these comments are short or incomplete. For example, a method named `format`, which formats and appends various types of `units` to a string, is documented with the one-word comment `Formatting`. As a result, relying solely on instances with sufficiently detailed developer-written comments would have produced a dataset that is both small and inconsistent. To address this, we adopted the opposite approach: We retained only the code from the instances in D_b and used an automated methodology to generate comments, creating the alternative dataset with comments through automated comment generation.

Table 2: Comparing the old and renewed datasets in terms of number of tokens and cyclomatic complexity (CC) for both the *buggy* (B) and *fixed* (F) parts of the instances.

Dataset	# Tokens (B)	# Tokens (F)	CC (B)	CC (F)
BFP_{small}	31.92	29.16	1.35	1.36
BFP_{medium}	75.07	73.68	2.45	2.50
Renewed (Our)	101.29	108.36	3.22	3.49

As a preliminarily step, we removed all the developer-written comments from the methods in D_b. Then, we relied on GPT-3.5 [6] to generate multi-intent comments for a method to document with an in-context learning setting [6], similarly to Geng *et al.* [18]. The authors showed that providing examples in the prompts helps the models to generate comments that address the different aspects of the method (Section 3.2). We relied on GPT-3.5 to balance the performance of GPT-based models with a manageable cost and because Su and McMillan [53] found that it generates code summaries that are preferred by developers even more than those used as 'ground-truth' in existing datasets. Based on our theory reported in Section 3, we generate comments for such intents (*i.e., what*, *how*, *why*, *usage*, and *property*). We use a prompt template with 2 examples (two-shot learning) that provides GPT-3.5 with sufficient information to generate the comments. We explicitly ask the model to generate a single sentence comment for each category for two reasons. First, this choice aligns with code-summarization research, in which the objective is, typically, to generate a single sentence summary [2, 21, 52, 53]. Second, we want to avoid an overly unrealistic scenario in which comments exceed code (which would have happened for smaller methods if we did not impose this constraint). The prompt template we adopted is reported below:

> You are an expert software developer and you are great at writing comments as requested. Your role is to write a single sentence of comment for each one of the following categories: what, why, how-it-is-done, how-to-use, property. Below you find some examples: `{example-1}`; `{example-2}`. Now write the comment for the following Java method: `{target-method}`.

The two examples (`{example-1}` and `{example-2}`) are reported in Fig. 2. We choose the two explicit examples of multi-intent comment generation from the replication package of the work by Mu *et al.* [44]. One of them is a trivial example (first one in Fig. 2), while the second one is more complex (second one in Fig. 2) to expose the model to different levels of challenge. The only part of the whole prompt template that changes for each instance is `{target-method}`. Given an instance $\langle m_i^{buggy}, m_i^{fixed} \rangle$ in D_b, we replace `{target-method}` with m_i^{fixed}, *i.e.,* the *fixed* version of the method. We generate comments for the *fixed* version rather than for the *buggy* version because the latter code does not completely meet the requirements (by definition). The comments generated from it might reflect such errors. The presence of wrong comments might not allow us to test our theory. For each instance $\langle m_i^{buggy}, m_i^{fixed} \rangle$ in D_b, we define a comment-augmented alternative by pre-pending the generating comments to both the *buggy* and *fixed* versions, leading to $\langle CG(m_i^{fixed}) + m_i^{buggy}, CG(m_i^{fixed}) + m_i^{fixed} \rangle$, where CG is the comment generation methodology we previously defined. This leads to the definition of our comment-augmented alternative of D_b, which we call D_c. We did not tune the prompt to optimize the quality of the generated comments since proposing a comment generation approach was beyond the scope of our work.

It is worth noting that using the fixed version of the code to generate comments makes this methodology inapplicable as-is for improving ABF in practical scenarios, as the *fixed* version is obviously not available beforehand. However, our objective is not to propose a practical methodology for enhancing bug-fixing but to validate our theory and determine whether comments play a significant role in the first place.

Validating Generated Comments. We conducted a manual analysis on a random sample including 400 instances to validate the generated comments. Specifically, we evaluated the *coherence* [44], defined as whether a comment correctly reflects its intended category (*e.g.,* what, why, how), and the *adequacy* [41], defined as the extent to which the comment correctly describes those aspects of the source code and does not make mistakes. Coherence is assessed in a binary way (*coherent* or *not coherent*), while adequacy is assessed on a 1 to 5 Likert scale. Two of the authors independently performed such an analysis and discussed any disagreement trying to reach consensus. As for adequacy, we considered two independent evaluations as not in agreement if one of them was positive (4 or 5) and the other one was negative (1 or 2). In the other cases, we computed the overall adequacy as the mean of the two assigned scores. In the *coherence* analysis, the evaluators initially agreed on 349 instances (87% agreement), while the remaining 51 were carefully reviewed and resolved in a meeting. In contrast, no disagreements arose in the *adequacy* analysis, for which we observed a substantial agreement (Cohen's κ [12] = 0.72). We observed that 93% of the analyzed comments are coherent with the related comment category and that the median adequacy score assigned is equal to 4. Therefore, we can claim that the generated comments are sufficiently good for our experiment.

Final Filtering and Splitting. The comment-augmented dataset D_c naturally contains larger instances than D_b (it contains the same code plus the generated comments). For this reason, some of the instances in D_c contain more than 512 tokens (the maximum our model can handle). Thus, we remove from D_c the instances that contain more than 512 tokens. We remove such instances also from D_b, ensuring that we still have the same methods in both the datasets. In this step, we remove 6,163 instances from both datasets. We also discarded from both the dataset all the instances for which GPT-3.5 was not able to correctly generate the comments (110). We identified such instances by automatically verifying, via regular expressions, that the generated comments followed the expected format, composed all the types (*e.g.,* `what:`) followed by a sentence.

We found those instances by filtering out those that did not respect the expected comments structure (*i.e.,* "what: what-comment", etc.). We randomly partition the instances in D_b into three sets (fine-tuning Φ_b, test T_b, and validation V_b) following the 80-10-10 rule, like most previous work did [5, 39, 40, 62]. We then split D_c into fine-tuning, test and validation sets (Φ_c, T_c, and V_c, respectively) by making sure that each instance $i_c \in D_c$ was assigned to the same type of set as D_b. For example, if an instance $i_b \in D_b$ was assigned to the fine-tuning set Φ_b, its commented counterpart, $i_c \in D_c$ was assigned to the fine-tuning set Φ_c. As a result, we reserved 93,098 pairs for the fine-tuning sets (Φ_b and Φ_c), 11,637 for the test sets (T_b and T_c), and 11,637 for for the validation (V_b and V_c).

4.2 Experimental Procedure

RQ$_1$: Comments Impact. To answer RQ$_1$, we conduct a *controlled experiment* [67]. The experimental *objects* are both (i) the fine-tuned models, and (ii) the test sets with and without comments we previously defined. We identified several independent variables (IVs) related to the model training and configuration that might affect the dependent variables (DVs), which are related to the effectiveness of the model. We manipulate some of them (the *factors*, *i.e.,* the variables for which we want to study the effect on the DVs). We describe below in details the variables and the procedure we used to conduct the experiment.

We use two LLMs in our experiment: CodeT5+ [64] and DeepSeek-Coder [20]. As for CodeT5+, we employ the smallest variant with 220M parameters, using an encoder-decoder architecture suitable for the bug-fixing task [5, 39, 56]. As for DeepSeek-Coder, we employ the smallest instruction variant with 1.3B parameters, which features a decoder-only architecture. The first independent variable (IV_{LLM}) takes two values: CodeT5+ and DeepSeek-Coder. We choose to leverage such models since (i) they are specifically pre-trained to handle coding tasks, (ii) they have been extensively used in previous work [7, 27, 63, 66], and (iii) they employ different model architectures (*i.e.,* encoder-decoder and decoder-only). For each LLM, we fine-tune two versions of the model using different datasets: one of them (Φ_c) contains instances with the comments generated with the previously explained procedure ($IV_{Comments_{train}}$ = *Yes*), and the other one (Φ_b) contains the same instances without any comment ($IV_{Comments_{train}}$ = *No*). Formally, the independent variable $IV_{Comments_{train}}$, which represents the presence of comments in fine-tuning instances, is the first *factor* in our experiment, for which we have the two *treatments*, *i.e., Yes* (*all* the code instances have a comment) and *No* (*none* of the code instances has comments). We call the models fine-tuned on the datasets $M_{\Phi_c}^{CT}$, $M_{\Phi_b}^{CT}$ (CodeT5+), and $M_{\Phi_c}^{DS}$, $M_{\Phi_b}^{DS}$ (DeepSeek-Coder), respectively. The fine-tuning process differs between the models. For CodeT5+ the number of

```
Code:
public int hashCode() {
    return value.hashCode();
}

Comment:
what: Generates a hash code.
why: Generates code for this object.
how: A method that generates a hash code based on the contents of the string representations.
usage: This method is used when this class is used as the code.
property: Return a hashcode for this text attribute.

Code:
<T> List<T> onFind(Class<T> modelClass, String[] columns, String[] conditions, String orderBy, String limit, boolean isEager) {
    BaseUtility.checkConditionsCorrect(conditions);
    List<T> dataList = query(modelClass, columns, getWhereClause(conditions), getWhereArgs(conditions), null, null,
                             orderBy, limit, getForeignKeyAssociations(modelClass.getName(), isEager));

    return dataList;
}

Comment:
what: Handles the native query of the given table.
why: The open interface for other classes in CRUD package to query the first record in a table.
how: Finds generic values by the conditions specified in the backwards conditions object, the where clause is the product of each of the type.
usage: Method modified just override here before executing your test cases.
property: Returns the order of the order specified.
```

Figure 2: The two examples we use for few-shot learning in our prompt.

epochs is fixed to 50 (IV_{Epochs} = 50), the batch size is 12 (IV_{BS} = 12), the optimizer is AdamW (IV_{opt} = AdamW), and the input and output size is limited to a maximum of 512 tokens (IV_{input} = IV_{output} = 512). For DeepSeek-Coder, the number of epochs is fixed to 10 (IV_{Epochs} = 10), the batch size is 32 (IV_{BS} = 32), and the optimizer is AdamW (IV_{opt} = AdamW). Differently from encoder-decoder models, decoder-only models processes the input and generated output tokens as a single continuous sequence during generation. For this reason, we fixed a combined sequence length limit of 2048 tokens (IV_{max_length} = 2048), ensuring that the total number of input and output tokens does not exceed this value. We employ early stopping (IV_{ES} = *Yes*) to prevent overfitting [48]. Specifically, we evaluate the models after each epoch on the respective validation sets (V_c for $M^{CT}_{\Phi_c}$ and $M^{DS}_{\Phi_c}$, and V_b for $M^{CT}_{\Phi_b}$ and $M^{DS}_{\Phi_b}$) and take the best checkpoint before the accuracy decreases for five consecutive epochs (patience).

We evaluate each fine-tuned model on both types of test sets to study the impact of comments during both fine-tuning and inference. This corresponds to the independent variable $IV_{Comments_{test}}$, which controls the presence of comments in the test instances (*Yes*/*No*). Specifically, $M^{CT}_{\Phi_b}$ and $M^{DS}_{\Phi_b}$ (trained without comments) are tested on both T_b ($IV_{Comments_{test}}$=*No*) and T_c ($IV_{Comments_{test}}$=*Yes*), and $M^{CT}_{\Phi_c}$ and $M^{DS}_{\Phi_c}$ (trained with comments) are also tested on both T_b and T_c. We use greedy decoding as decoding strategy, *i.e.,* the model generates the token with the highest probability at each generation step. We choose such a strategy rather than using sampling-based decoding methods to study the actual models learned behaviours during the fine-tuning process.

We consider two *dependent variables*: Exact Match (EM) and CodeBLEU (CB). EM represents the number of the correct inferences made by the model on the test set divided by the size of the test set. CB [51] measures the similarity between the inference made by the model and the expected generated code. Such a metric is a variation of BLEU [47] specialized on the source code. While BLEU relies on matching n-grams, CodeBLEU considers the syntax (through the Abstract Syntax Tree) and its semantics (via data-flow).

In addition, we perform statistical tests to compare the different training and inference configurations in terms of EM. We use McNemar's test [42] on paired test instances, along with the odds ratio (OR) to measure the magnitude of the effect size. We reject the null hypothesis (*i.e.,* no difference in effectiveness between two configurations) if the p-value is lower than 0.05. To interpret the magnitude of the effects, we categorize odds ratios according to Cohen's guidelines [4, 13]: *very small* (OR $<$ 1.44), *small* (1.44 $\leq$ OR $<$ 2.48), *medium* (2.48 $\leq$ OR $<$ 4.27), and *large* (OR $\geq$ 4.27).

RQ$_2$: Comments Importance. To answer RQ$_2$, we employ post-hoc interpretability techniques to understand to what extent tokens that are part of the comments are important in the inferences the models $M^{CT}_{\Phi_c}$ and $M^{DS}_{\Phi_c}$ make. We use SHAP (SHapley Additive exPlanations) [33], a black-box model-agnostic method. As previously explained, LLMs (and thus our models) generate one token at a time. Given a token t_k that the model has to generate from the given context χ_{t_k}, related to the instance $i_c \in T_c$, SHAP assigns a *feature importance* value f to each token $t \in \chi_{t_k}$. For our analyses, we use the same methodology employed by Liu *et al.* [32], *i.e.,* we adopt GradientExplainer to approximates SHAP values. More specifically, for each instance of the test set $\langle m_i^{buggy}, m_i^{fixed} \rangle \in T_c$ that the model correctly predicts, we compute the f values of all the tokens in m_i^{buggy} for each token that the model generates. We ignore the f values assigned to the tokens that the model progressively generates since we are interested in the role of the input code. We normalize the SHAP values so that their sum is equal to 1.

In a first analysis, we aimed to understand the importance given to the comments as compared to the importance given to the source

Table 3: Comparison between the models trained and tested with and without comments.

Models	T_b		T_c	
	EM	CB	EM	CB
$M^{CT}_{\Phi_b}$ (*Comments* = No)	3.26%	76.21%	3.73%	75.66%
$M^{CT}_{\Phi_c}$ (*Comments* = Yes)	2.78%	78.18%	9.80%	80.38%
$M^{DS}_{\Phi_b}$ (*Comments* = No)	5.57%	77.72%	7.90%	86.40%
$M^{DS}_{\Phi_c}$ (*Comments* = Yes)	5.09%	77.37%	12.77%	81.01%

code. For each instance $i_c \in T_c$, we partition the tokens in two sets: tokens belonging to the comments (C_{i_c}) and tokens belonging to the source code (S_{i_c}). We then estimate the importance of tokens in the comments ($I_{Comments}(i_c)$) by summing the f-values assigned to the tokens in C_{i_c}, and the importance of tokens in the code $I_{Code}(i_c)$ by summing the f-values assigned to the tokens in S_{i_c}. We report the distribution of $I_{Comments}(i_c)$ and $I_{Code}(i_c)$ for all the instances. In a second more fine-grained analysis, we aimed to understand what is the importance assigned to the five categories of comments we generated for each instance. To do this, for each instance $i_c \in T_c$, we focused on the tokens belonging to the comments (C_{i_c}) and normalized the f-values computed on them so that their sum is equal to 1. We partitioned the tokens in five classes based on the specific comment they belonged to (*what*, *why*, *how*, *usage*, and *property*). We could easily do this because the comments were always generated in the same order with our prompt (*e.g.,* the first comment was always aimed at explaining *what* the method does). We computed the importance for each of them, again, by summing the f-values. We report the distribution of $I_{What}(i_c)$, $I_{Why}(i_c)$, $I_{How}(i_c)$, $I_{Usage}(i_c)$, and $I_{Property}(i_c)$ for all the instances.

5 Empirical Study Results

We report the results of our empirical study and the answers to our two RQs.

5.1 RQ$_1$: Comments Impact

The results obtained in terms of Exact Match (EM) and CodeBLEU (CB) for the four fine-tuned models—$M^{CT}_{\Phi_b}$ and $M^{DS}_{\Phi_b}$ (trained without comments), and $M^{CT}_{\Phi_c}$ and $M^{DS}_{\Phi_c}$ (trained with comments)—evaluated on the two test sets T_b (without comments) and T_c (with comments) are summarized in Table 3. All differences discussed in the following are statistically significant according to McNemar's test (p-value < 0.05). This is due to the large size of the test set. Therefore, we focus the discussion on the effect size, as measured by the odds ratio (OR).

We start by analyzing the results on the test set T_b, where no comments are provided to the model at inference time. Under this setting, the CodeT5+ model trained without comments ($M^{CT}_{\Phi_b}$) achieves an EM of 3.26% and a CB of 76.21%. When trained with commented instances ($M^{CT}_{\Phi_c}$), the model reaches an EM of 2.78% and a slightly higher CB of 78.18%. This difference, while statistically significant, reports a *very small* effect size (OR = 0.72), indicating a negligible practical impact. A similar trend is observed for DeepSeek-Coder: the model without comments in training ($M^{DS}_{\Phi_b}$) achieves an EM of 5.57% and CB of 77.72%, while the model trained with comments ($M^{DS}_{\Phi_c}$) reports an EM of 5.09% and CB of 77.37%. Such a comparison corresponds to a *very small* effect size (OR = 0.65). These results indicate that, when comments are absent at test time, models trained with comments achieve slightly lower performance than those trained without them. Surprisingly, this behavior suggests that the presence of comments during training does not introduce harmful biases but leads to only slightly weaker performance in contexts lacking comments, likely because such models tend to rely on contextual information that are absent at test time.

> Training with comments does not significantly impact performance in comments-free settings.

When comments are included at test time (T_c), both model families benefit substantially. For CodeT5+, the non-commented variant ($M^{CT}_{\Phi_b}$) increases its EM from 3.26% to 3.73% (OR = 1.44), showing that comments in the input alone can provide marginal guidance even without prior exposure. However, the most evident result emerges for the model trained with comments ($M^{CT}_{\Phi_c}$), whose EM increase from 2.78% to 9.80% (*i.e.,* ∼3× higher) and CB from 78.18% to 80.38%, with a *large* effect size (OR = 6.86). A similar, and more pronounced pattern appears for DeepSeek-Coder: $M^{DS}_{\Phi_b}$ improves from 5.57% to 7.90% EM when tested on commented code (*medium* effect size, OR = 3.42), while $M^{DS}_{\Phi_c}$ achieves the best overall results, with an EM of 12.77% and CB of 81.01%, approximately ∼2× the previous 5.09% with a *large* effect size (OR = 9.76).

These results highlight two main behaviors. First, models trained without comments still benefit from comments at inference time, fixing more bugs when contextual information is available. Second, models trained with comments *significantly* benefit from such contextual information, reporting performance improvements of approximately two to three times when comments are provided at test time. In both model families, the combination of training and testing with comments results in the largest performance, showing that comments act as useful auxiliary signals that enhance the models reasoning and code comprehension abilities.

> Comments consistently improve model performance, regardless of the training setup. The improvements are more evident when the model is trained to also leverage such information.

Across all configurations, DeepSeek-Coder consistently outperforms CodeT5+, likely due to the larger number of parameters, different architecture, and wider pre-training corpus. This confirms that larger models generally possess stronger bug-fixing capabilities. Interestingly, the positive effect of adding comments at inference time, when models have not been trained with them, is more evident for the larger DeepSeek-Coder, suggesting that bigger models can better exploit contextual information even without previous exposure. On the other hand, when fine-tuning is performed also with commented instances, the smaller CodeT5+ benefits the most, showing a three times EM improvement when tested with comments, compared to approximately two times improvements for DeepSeek-Coder. This indicates that while larger models are naturally more capable, smaller ones can close much of the gap when guided by explicit contextual information such as comments.

Answer to RQ$_1$. Comments enhance LLM bug-fixing by providing contextual information that improves performance even when available only at inference time. Training with comments remains neutral in comment-free cases but highly beneficial when comments are present during inference.

5.2 RQ$_2$: Comments Importance

Fig. 3 illustrates the importance attributed by the models $M^{CT}_{\Phi_c}$ and $M^{DS}_{\Phi_c}$ to tokens belonging to code and comments for their respective 1,141 and 1,486 exact matches. We find that the models consistently assign higher importance to code-related tokens (*i.e.,* those representing the logic of the method) in nearly all instances, with fewer than 100 exceptions for both models. However, the boxplot distributions reveal distinct patterns: for $M^{CT}_{\Phi_c}$, the importance is nearly evenly distributed between code and comments, whereas $M^{DS}_{\Phi_c}$ assigns greater importance to code tokens while still recognizing the value of comment tokens. This analysis, from a different perspective, supports the findings from RQ$_1$: Comments contribute significantly to the fix generation process.

When we focus on the different comment categories (right part of Fig. 3), we find that some comments are more important than others. Both models give much higher importance to comments describing *how* the implementation is done as compared to the other categories. This is quite expected: Such comments document some implementation details that might have been violated in the *buggy* version. Thus, the models are able to find the violation and fix it thanks to them. Then, we find that the $M^{CT}_{\Phi_c}$ gives similar importance to the *property*-related comments, the ones describing the method *usage*, and its design rationale (*why*). On the other hand, it gives less importance to the comments aimed at describing *what* the method does. We conjecture that this happens for two reasons. First, there are generally other elements that help the model grasp *what* the method is aimed to do (*e.g.,* the method name or the identifiers). Such elements make such a comment less relevant than others. Second, it is probably more rare that a bug makes the method do something different from what is supposed to do, so that a comment describing what the method does helps the model fixing it. Regarding $M^{DS}_{\Phi_c}$, we find that the model gives lower importance attributions to *property*-related and *usage*-related comments, while giving higher importance to *why* and *what* comments categories. The models give contrasting importance to *what* comments. We conjecture that the different pre-training objectives employed between the models may impact on such findings [20, 64]. In particular, the base DeepSeek-coder model was pre-trained using a next-token prediction objective. For this reason, instances used during pre-training might have included inputs that better align with the high-level descriptions generally found in *what* comments.

Answer to RQ$_2$. Code comments are assigned significant importance by the models. The most important comments are those documenting implementation details, while the least important are *what* comments for $M^{CT}_{\Phi_c}$ and *property* comments for $M^{DS}_{\Phi_c}$.

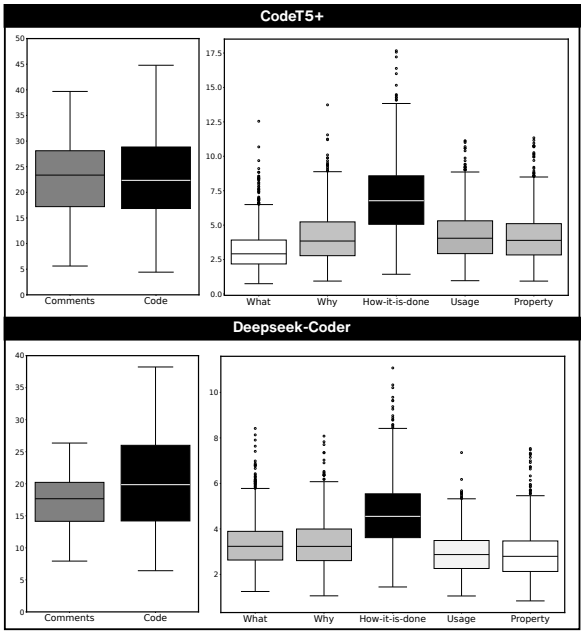


Figure 3: Importance distribution for comments and code (left part) and comment intents (right part).

6 Discussions and Implications

In this section, we discuss the lessons learned and implications for future research based on the results of our controlled experiment.

6.1 Lessons Learned

Our results clearly show that comments are beneficial to improve the effectiveness of LLMs in the ABF task. While this is true in most cases, we found some instances for the generated comments are not sufficiently detailed to fix the bug. In Fig. 4 we see an interesting example. In the buggy version, the method attempted to set an answer "`id`" in the "`SaveAnswerEvent`" object, which could result in incomplete event data if an exception occurred during the save operation. The fix removed such an assignment, ensuring the event is posted without depending on the successful retrieval of the `id`. However, the model predicted a candidate fix that set the "`answer`" object in the event, leading to an analogous problem. It probably did that because the input code contained a call to the `setAnswerId` method and the model did not try to change much the code structure. In addition, the *how* comment states *"... and catches any exceptions to set in the event."*, which could mislead the model. We tried to change the comment by removing such a part of the comment, and this resulted in a correct fix.

On the other hand, there are cases in which generated comments are sufficient to help the model generate the correct fix. As we can see in Fig. 5, the method did not store the `aggregationBuffer` in the `hashMap`. The *property*-related comment most likely helped fix this bug since it explicitly says that the method *ensures that an aggregation buffer is available for the specified grouping key*, property that is violated in the buggy version of the code. The model without comments fails to correctly fix the method by using `aggregationBufferFactory` instead of storing the created `SpecificMutableRow` in the `hashMap`.

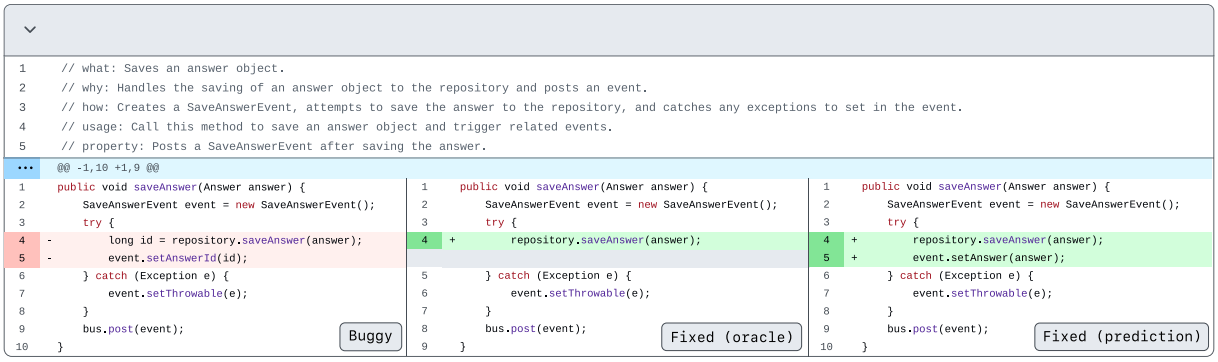


Figure 4: Example of wrong prediction.

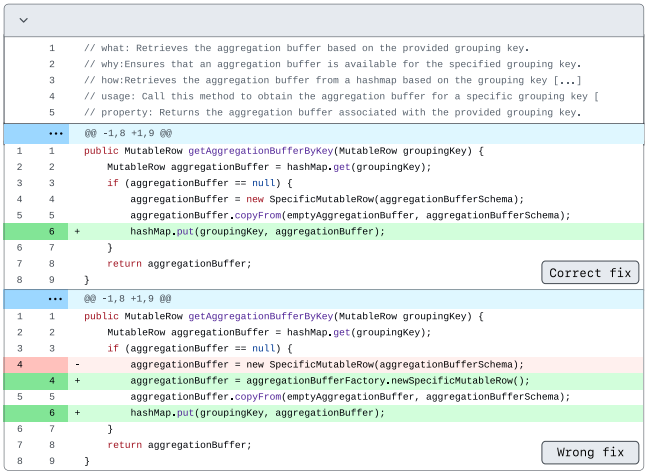


Figure 5: Example of a $M^{CT}_{\Phi_c}$ correct fix, and $M^{CT}_{\Phi_b}$ wrong fix.

Lesson Learned 1. Although comments can boost the performance of automated bug-fixing models, their completeness and correctness is key to fix bugs.

In a real use-case scenario, however, it is unlikely that developers provide comments for all the categories we studied. Most importantly, developers might be reticent to provide comments regarding the implementation details because the such details might change frequently, thus leading to inconsistencies between comments and code and an increased effort. All the other categories of comments, instead, might change less frequently. To test what would happen in a more realistic context, we fine-tuned and evaluated $M^{CT}_{\Phi_{c*}}$ on a variation of the D_c dataset (D_{c*}), from which we removed the *how* comments, and tested it on the corresponding test set T_{c*}, where the *how* comments were also removed. Even without implementation details, we still observe a significant boost in terms of EM, which is 91% higher than the baseline model ($M^{CT}_{\Phi_b}$ on T_b), with higher CodeBLEU. This result shows that even if not too detailed, comments can help improving ABF.

Lesson Learned 2. Even non-implementation related comments can help to improve the effectiveness of LLMs for ABF.

We further explored what would happen if the comments were instead generated from the *buggy* version of the code. This scenario represents a realistic context, in which comments can be automatically generated or manually written based on the available buggy code. However, this setting *inherently* implies incorrect comments, since they document the buggy logic rather than the intended correct behavior. Thus, we tested the previously fine-tuned models on a variant of the test set with comments generated from the buggy code. As expected, all models performed the same or slightly worse with buggy-generated comments. For instance, CodeT5+ saw small EM drops (3.26% to 2.92%, 2.78% to 2.67%), with similar negligible or negative trends for DeepSeek-Coder. These results confirm that comments generated from buggy code do not provide useful guidance and can even hinder models, since they reinforce incorrect behaviors embedded in the buggy code.

Lesson Learned 3. Comments generated from buggy code do not help and can mislead automated bug-fixing models.

6.2 Implications

Our findings have several implication for both researchers and practitioners. First, removing code comments from training instances is a common practice in ABF research [56]. However, given the results of our findings we suggest researchers to not remove comments from training instances. Low-quality comments should be improved, whenever possible. Moreover, researchers could devise approaches to automatically generate high-quality comments that serve as effective contextual information, inferred from the broader project context. For practitioners, our findings highlight the importance of thoroughly documenting code to better leverage the capabilities of LLMs for automating bug-fixing. While it is widely recognized that code documentation improves maintainability, our results suggest that its benefits extend beyond internal quality, positively impacting external quality by enhancing the support developers receive in the bug fixing process by LLMs. Although our study focused on ABF, we conjecture that these findings may apply to other tasks as well, which future research should aim to confirm or refute. Finally, tool builders could develop features to identify comments lacking

sufficient detail (*e.g.,* those that fail to explain rationale, preconditions, or postconditions) or of low quality (*e.g.,* inconsistent with the code). Such tools could alert developers when attempting to use LLMs to repair methods with inadequate comments and encourage improvements, potentially supported by automated suggestions.

7 Threats to Validity

Threats to construct validity. We started from the dataset provided by Tufano *et al.* [56] to define the datasets we employ in our study, *i.e.,* D_b and D_c. As the authors themselves acknowledged, some instances might not represent actual bug-fixing instances or the bug-fixing change might be in a method different from the one in a given instance. To limit this threat, we used a stricter criterion to select instances, *i.e.,* we only considered commits impacting a single method. We defined D_c leveraging GPT-3.5 to generate the multi-intent comments. It is possible that some generated comments are not appropriate for a given category (*e.g., usage*). Besides, it is possible that such comments are not good enough. To limit this threat, we conducted a manual analysis on random sample including 400 generated comments. We evaluated the *coherence* to the given category (*coherent* or *not coherent*) and the *adequacy* to the source code (on a 1 to 5 Likert scale). Two of the authors independently performed such an analysis. If any disagreement arose, they would have discussed trying to reach a consensus. It is also possible that some instances in our dataset appeared in the datasets used to pre-train CodeT5+ and DeepSeek-Coder. We started from the same pre-trained models to define the specialized bug-fixing models trained with and without comments. If such an issue biased the results, it did so for both the models. Thus, we believe that the *relative* results (*i.e.,* the difference between the models) was not affected by this possible issue whatsoever.

Threats to internal validity. We used EM and CB as dependent variables of our experiment to measure the effectiveness of the models. EM is a lower bound of the actual bug-fixing capabilities of the models we tested. A prediction that exactly matches the expected output is surely a bug fix; on the other hand, a prediction not exactly matching the excepted output is not necessarily a wrong fix as it could contain a valid alternative patch. For example, if the expected fix contains a call to `Collections.isEmpty(list)` while the predicted one contains `list.isEmpty()`, the latter is a valid bug fix even if not exactly matching the expected one. Thus, it is possible that the *absolute* capabilities of the models are actually higher. We use CodeBLEU to measure the distance between the predicted sequence and the expected one. We used this as an estimate of the effort a developer would make to transform an incorrect fix provided by the model into the correct fix. It is worth noting, however, that a small number of tokens to change does not necessarily imply that developers would make a smaller effort. Finally, we adopted a state-of-the-art post-hoc interpretability methods [32, 33] to estimate the importance given by the model to comments (in general) and to different categories of comments (specifically). It is worth noting that such an approach, like others available, only provide estimates on the importance given by the model to single tokens. Thus, it is possible that the importance that the model assigns to comments and to different comment categories differs from the one we report. It is worth noting, however, that when removing the supposedly best category of comments according to our analysis (*i.e., how* comments) we obtain lower EM than the one obtained when using all the comments (even if still higher than the one obtained without comments). While this suggest that the reported values are sufficiently reliable, further tests with single categories of comments are required to confirm this result.

Threats to external validity. The dataset by Tufano *et al.* [56] from which we built our dataset contains instances relatively old and limited to the Java programming language. Our results might not generalize to bug-fixing commits performed more recently and on different programming languages. It is worth noting that the dataset by Tufano *et al.* has been used in recent work as well [3, 8, 14, 19, 22–24, 30, 37, 39, 63, 65, 71]. Our results might not generalize to other models. To limit this threat, we tested our theory on two models, CodeT5+ [64] and DeepSeek-Coder [20]. The first representative of an encoder-decoder architecture and limited number of parameters (*i.e.,* 220M), and the second one representative of a decoder-only architecture with an higher number of parameters (*i.e.,* 1.3B). To further assess the generalizability of our findings, we evaluated GPT-4.1 in a zero-shot setting on the same test sets with and without comments, observing an increase in EM from 6.11% (without comments) to 8.63% (with comments). This supplementary result provides preliminary evidence that the positive impact of comments on automated bug fixing extends to more recent and larger LLMs. Still, we acknowledge that further empirical evidence is needed to confirm this phenomenon on different models and different configurations.

8 Conclusion

We conducted an empirical study to investigate the impact of comments on the Automated Bug Fixing (ABF) task. We compared models trained and tested on all combinations of bug-fix pairs with and without automatically generated, multi-intent comments focusing on five intents (*what*, *why*, *how*, *usage*, and *property*).

Our results show that the presence of comments—whether provided during training, testing, or both—substantially improves the effectiveness of large language models in fixing bugs, with accuracy improvements of up to three times compared to configurations where comments are absent. When analyzing the importance of the five categories of comments we took into account, we found that comments that explain *how* the method is implemented are the most important ones, while the ones related to *what* and/or *property* the method does are less relevant.

Future studies are needed to provide further empirical evidence in support of our findings, also on other tasks (*e.g.,* vulnerability fixing). Besides, it will be interesting to study to what extent the comments provided by developers are sufficient to improve the effectiveness in bug fixing.

Acknowledgments

This publication is part of the project PNRR-NGEU which has received funding from the MUR – DM 118/2023. This work has been partially supported by the European Union - NextGenerationEU through the Italian Ministry of University and Research, Projects PRIN 2022 "QualAI: Continuous Quality Improvement of AI-based Systems", grant n. 2022B3BP5S, CUP: H53D23003510006.

References

[1] 2025. Replication Package for "On the Impact of Code Comments for Automated Bug-Fixing: An Empirical Study". doi:10.5281/zenodo.17408250

[2] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. *arXiv preprint arXiv:2005.00653* (2020).

[3] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333* (2021).

[4] Mattan S. Ben-Shachar, Daniel Lüdecke, and Dominique Makowski. 2020. effectsize: Estimation of Effect Size Indices and Standardized Parameters. *Journal of Open Source Software* 5, 56 (2020), 2815. doi:10.21105/joss.02815

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

[6] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[7] Federico Cassano, Luisa Li, Akul Sethi, Noah Shinn, Abby Brennan-Jones, Jacob Ginesin, Edward Berman, George Chakhnashvili, Anton Lozhkov, Carolyn Jane Anderson, et al. 2023. Can it edit? evaluating the ability of large language models to follow code editing instructions. *arXiv preprint arXiv:2312.12450* (2023).

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

[9] Qiuyuan Chen, Xin Xia, Han Hu, David Lo, and Shanping Li. 2021. Why my code summarization model does not work: Code comment improvement with category prediction. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 2 (2021), 1–29.

[10] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering* 47, 9 (2019), 1943–1959.

[11] Matteo Ciniselli, Nathan Cooper, Luca Pascarella, Antonio Mastropaolo, Emad Aghajani, Denys Poshyvanyk, Massimiliano Di Penta, and Gabriele Bavota. 2021. An empirical study on the usage of transformer models for code completion. *IEEE Transactions on Software Engineering* 48, 12 (2021), 4818–4837.

[12] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and psychological measurement* 20, 1 (1960), 37–46.

[13] Jacob Cohen. 2013. *Statistical power analysis for the behavioral sciences*. routledge.

[14] Dawn Drain, Chen Wu, Alexey Svyatkovskiy, and Neel Sundaresan. 2021. Generating bug-fixes using pretrained transformers. In *Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming*. 1–8.

[15] Hadeel Eladawy, Claire Le Goues, and Yuriy Brun. 2024. Automated Program Repair, What Is It Good For? Not Absolutely Nothing!. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[16] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 313–324.

[17] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999* (2022).

[18] Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Ge Li, Zhi Jin, Xiaoguang Mao, and Xiangke Liao. 2024. Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.

[19] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366* (2020).

[20] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming–The Rise of Code Intelligence. *arXiv preprint arXiv:2401.14196* (2024).

[21] Sakib Haque, Alexander LeClair, Lingfei Wu, and Collin McMillan. 2020. Improved automatic summarization of subroutines via attention to file context. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 300–310.

[22] Soneya Binta Hossain, Nan Jiang, Qiang Zhou, Xiaopeng Li, Wen-Hao Chiang, Yingjun Lyu, Hoan Nguyen, and Omer Tripp. 2024. A deep dive into large language models for automated bug localization and repair. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1471–1493.

[23] Kai Huang, Xiangxin Meng, Jian Zhang, Yang Liu, Wenjie Wang, Shuhao Li, and Yuqing Zhang. 2023. An empirical study on fine-tuning large language models of code for automated program repair. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1162–1174.

[24] Kai Huang, Jian Zhang, Xinlei Bao, Xu Wang, and Yang Liu. 2025. Comprehensive Fine-Tuning Large Language Models of Code for Automated Program Repair. *IEEE Transactions on Software Engineering* (2025).

[25] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of code language models on automated program repair. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1430–1442.

[26] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. Cure: Code-aware neural machine translation for automatic program repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1161–1173.

[27] Kisub Kim, Jounghoon Kim, Byeongjo Park, Dongsun Kim, Chun Yong Chong, Yuan Wang, Tiezhu Sun, Daniel Tang, Jacques Klein, and Tegawendé F Bissyandé. 2024. DataRecipe—How to Cook the Data for CodeLLM?. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1206–1218.

[28] Taku Kudo and John Richardson. 2018. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. *arXiv preprint arXiv:1808.06226* (2018).

[29] Alexander LeClair and Collin McMillan. 2019. Recommendations for datasets for source code summarization. *arXiv preprint arXiv:1904.02660* (2019).

[30] Bo Lin, Shangwen Wang, Ming Wen, Liqian Chen, and Xiaoguang Mao. 2024. One size does not fit all: Multi-granularity patch generation for better automated program repair. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1554–1566.

[31] Fang Liu, Ge Li, Zhiyi Fu, Shuai Lu, Yiyang Hao, and Zhi Jin. 2022. Learning to recommend method names with global context. In *Proceedings of the 44th International Conference on Software Engineering*. 1294–1306.

[32] Yue Liu, Chakkrit Tantithamthavorn, Yonghui Liu, and Li Li. 2024. On the reliability and explainability of language models for program generation. *ACM Transactions on Software Engineering and Methodology* 33, 5 (2024), 1–26.

[33] Scott M Lundberg and Su-In Lee. 2017. A unified approach to interpreting model predictions. *Advances in neural information processing systems* 30 (2017).

[34] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. 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*. 101–114.

[35] Antonio Mastropaolo, Emad Aghajani, Luca Pascarella, and Gabriele Bavota. 2021. An empirical study on code comment completion. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 159–170.

[36] Antonio Mastropaolo, Matteo Ciniselli, Luca Pascarella, Rosalia Tufano, Emad Aghajani, and Gabriele Bavota. 2024. Towards Summarizing Code Snippets Using Pre-Trained Transformers. In *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*. 1–12.

[37] Antonio Mastropaolo, Nathan Cooper, David Nader Palacio, Simone Scalabrino, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. 2022. Using transfer learning for code-related tasks. *IEEE Transactions on Software Engineering* 49, 4 (2022), 1580–1598.

[38] Antonio Mastropaolo, Luca Pascarella, and Gabriele Bavota. 2022. Using deep learning to generate complete log statements. In *Proceedings of the 44th International Conference on Software Engineering*. 2279–2290.

[39] Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. 2021. 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, 336–347.

[40] Antonio Mastropaolo, Fiorella Zampetti, Gabriele Bavota, and Massimiliano Di Penta. 2024. Toward Automatically Completing GitHub Workflows. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–12.

[41] Paul W McBurney and Collin McMillan. 2014. Automatic documentation generation via source code summarization of method context. In *Proceedings of the 22nd International Conference on Program Comprehension*. 279–290.

[42] Quinn McNemar. 1947. Note on the sampling error of the difference between correlated proportions or percentages. *Psychometrika* 12, 2 (1947), 153–157.

[43] Martin Monperrus, Matias Martinez, He Ye, Fernanda Madeiral, Thomas Durieux, and Zhongxing Yu. 2021. Megadiff: A dataset of 600k java source code changes categorized by diff size. *arXiv preprint arXiv:2108.04631* (2021).

[44] Fangwen Mu, Xiao Chen, Lin Shi, Song Wang, and Qing Wang. 2023. Developer-intent driven code comment generation. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 768–780.

[45] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022).

[46] Devon H O'Dell. 2017. The Debugging Mindset: Understanding the psychology of learning strategies leads to effective problem-solving skills. *Queue* 15, 1 (2017), 71–90.

[47] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 311–318.

[48] Lutz Prechelt. 2002. Early stopping-but when? In *Neural Networks: Tricks of the trade*. Springer, 55–69.

[49] Julian Aron Prenner and Romain Robbes. 2024. Out of context: How important is local context in neural program repair?. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[50] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research* 21, 140 (2020), 1–67.

[51] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297* (2020).

[52] Lin Shi, Fangwen Mu, Xiao Chen, Song Wang, Junjie Wang, Ye Yang, Ge Li, Xin Xia, and Qing Wang. 2022. Are we building on the rock? on the importance of data preprocessing for code summarization. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 107–119.

[53] Chia-Yi Su and Collin McMillan. 2024. Distilled GPT for source code summarization. *Automated Software Engineering* 31, 1 (2024), 22.

[54] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. 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*. 1433–1443.

[55] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).

[56] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28, 4 (2019), 1–29.

[57] Rosalia Tufano, Simone Masiero, Antonio Mastropaolo, Luca Pascarella, Denys Poshyvanyk, and Gabriele Bavota. 2022. Using pre-trained models to boost code review automation. In *Proceedings of the 44th international conference on software engineering*. 2291–2302.

[58] Rosalia Tufano, Luca Pascarella, and Gabriele Bavota. 2023. Automating code-related tasks through transformers: The impact of pre-training. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2425–2437.

[59] Tim van Dam, Maliheh Izadi, and Arie van Deursen. 2023. Enriching source code with contextual data for code completion models: An empirical study. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE, 170–182.

[60] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[61] Antonio Vitale, Rocco Oliveto, and Simone Scalabrino. 2024. A Catalog of Data Smells for Coding Tasks. *ACM Transactions on Software Engineering and Methodology* (2024).

[62] Antonio Vitale, Valentina Piantadosi, Simone Scalabrino, and Rocco Oliveto. 2023. Using Deep Learning to Automatically Improve Code Readability. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 573–584.

[63] Weishi Wang, Yue Wang, Shafiq Joty, and Steven CH Hoi. 2023. Rap-gen: Retrieval-augmented patch generation with codet5 for automatic program repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 146–158.

[64] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922* (2023).

[65] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).

[66] Martin Weyssow, Xin Zhou, Kisub Kim, David Lo, and Houari Sahraoui. 2023. Exploring parameter-efficient fine-tuning techniques for code generation with large language models. *arXiv preprint arXiv:2308.10462* (2023).

[67] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, Anders Wesslén, et al. 2012. *Experimentation in software engineering*. Vol. 236. Springer.

[68] He Ye, Matias Martinez, and Martin Monperrus. 2022. Neural program repair with execution-based backpropagation. In *Proceedings of the 44th international conference on software engineering*. 1506–1518.

[69] Juan Zhai, Xiangzhe Xu, Yu Shi, Guanhong Tao, Minxue Pan, Shiqing Ma, Lei Xu, Weifeng Zhang, Lin Tan, and Xiangyu Zhang. 2020. CPC: Automatically classifying and propagating natural language comments via program analysis. In *Proceedings of the ACM/IEEE 42nd International conference on software engineering*. 1359–1371.

[70] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A syntax-guided edit decoder for neural program repair. In *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*. 341–353.

[71] Armin Zirak and Hadi Hemmati. 2024. Improving automated program repair with domain adaptation. *ACM Transactions on Software Engineering and Methodology* 33, 3 (2024), 1–43.