Revisiting code readability improvement with LLMs: A critical assessment of fine-tuned models

Empirical Software Engineering (2027) 32:9
https://doi.org/10.1007/s10664-026-10933-0

Revisiting code readability improvement with LLMs: A critical assessment of fine-tuned models

Antonio Vitale[1] · Emanuela Guglielmi[2] · Valentina Piantadosi[2] · Antonio Mastropaolo[3] · Gabriele Bavota[4] · Rocco Oliveto[2] · Simone Scalabrino[2]

Received: 19 November 2025 / Accepted: 10 July 2026

Abstract

Code comprehension is vital for software development. Still, unreadable code remains a significant issue, costing substantial time and money losses. While tools to identify code exhibiting a low readability exist, take actions to improve such a quality aspect is far from trivial. To support developers in such a task, Vitale et al. introduced at ASE'23 an approach using a transformer model (T5) fine-tuned on code commits in which developers explicitly stated their goal to improve code readability. The authors reported that their model is able to generate readability-improving changes being identical to those implemented by developers (exact matches) in 21% of cases and that, even when they differ, they still improve readability in the vast majority of cases (∼80%). Given the major advances in AI made in the last few years, we questioned whether a fine-tuning for such a task was still needed in the era of Large Language Models (LLMs), thus revisiting the work by Vitale et al. with state-of-the-art models. In doing so, we found out one major issue with the original study design: The training-test splitting was performed randomly (*i.e.,* readability-improving commits mined from open source projects were randomly split between training and test) rather than by project, significantly inflating the reported performance due to repeating readability-improving commits done within the same project. Indeed, as we will show, even newer and more powerful LLMs fine-tuned for this task can only achieve a ∼1% of exact matches. Once fixed this issue, we shifted our attention on assessing whether fine-tuning on available developer readability improvements is actually needed given the recent advances in general-purpose LLMs. We compare the performance of fine-tuned state-of-the-art LLMs with what they achieve in zero-shot setting, also including in our study commercial LLMs (GPT-4.1). As we will show, quantitative metrics (*e.g.,* exact matches, CrystalBLEU) tell very little about the readability-improving capabilities of LLMs and, thus, our study is mainly qualitative in nature, with a total of ∼9,500 inspected LLMs' change recommendations. While all models do improve code readability, our results clearly show that fine-tuning using mined readability-improving commits does not help and, instead, results in sensibly poorer readability recommendations as compared to LLMs used in a zero-shot setting, with GPT-4.1 being the one achieving the best results.

Communicated by: Dan Hao.

Extended author information available on the last page of the article

Published online: 23 July 2026

Keywords Code readability · Large Language Models (LLMs)

1 Introduction

Software development hinges heavily on code comprehension, with such an activity consuming a substantial portion of developers' time (Erlikh 2000; Minelli et al. 2015). While this should make code readability as a first-class quality attribute, several software projects feature what has been defined in the literature as "unreadable code" (Piantadosi et al. 2020). Recognizing the criticality of this issue, prior research has focused on developing models to automatically assess code readability (Buse and Weimer 2008, 2009; Posnett et al. 2011; Dorn 2012; Scalabrino et al. 2016, 2018; Mi et al. 2018). While identifying unreadable code might make developers aware of what should be improved, it does not support them in solving the issue.

One might argue that readability can be easily improved by using existing IDE-integrated tools (*e.g.,* code formatting). However, this is only partially true, since code readability issues are often more intricate than they seem on the surface. Let us consider, as an example, the snippet in Fig. 1. There are several issues in the code *before* the implemented readability-improving change (see colored rectangles). The improvement of some of them is straightforward (*e.g.,* formatting issues). Others, instead, require more work and knowledge of good code design. For example, let us focus on line 3 *before* the change. When improving the naming of the variables (red rectangles) and removing magic numbers (blue rectangles), the line becomes too long, triggering an extract variable refactoring with *answerCount* being created. Finally, the method calls related to the Java stream APIs are split on several lines, again with the goal of improving readability.

While previous research has introduced advanced methods for improving all the individual aspects featured in this example (*e.g.,* styling Loriot et al. 2019 and naming Thies and Roth 2010; Allamanis et al. 2014; Lin et al. 2017), improving readability *as a whole* is a more complex problem to address. This is the reason why Vitale et al. (2023) recently proposed an approach that aims at achieving this goal. They first mine a dataset of readability-improving commits and then use it to fine-tune a T5 model (Raffel et al. 2019) that can improve code readability. Despite the pioneering nature of such a work, we wondered whether its findings were still valid nowadays, after the major advances done on general-purpose Large Language Models (LLMs), which can automate several tasks in a zero-shot setting (*i.e.,* by just receiving instructions via a crafted prompt). We started by replicating their work, exploiting the mining pipeline they built to enlarge the set of readability-improving commits mined from open source projects that are used to train and test the deep learning models. Indeed, Vitale et al. (2023) mined such commits from GitHub up to 2022 and, thus, additional data can be collected nowadays. Their pipeline exploits textual patterns in commit messages (*e.g.,* "*improve readability*") and has been shown to have a high accuracy in identifying readability-improving commits. As main quantitative evaluation metric in the original work, the authors used exact matches (EM), namely the percentage of commits in the test set for which T5 was able to recommend the exactly the same readability-improving changes as those implemented by developers. They found that, with a single attempt, T5 achieved EM=21%, a quite impressive result considering the very limited size (60M trainable parameters) of such a model. In this work, we exploit newer LLMs, since the T5

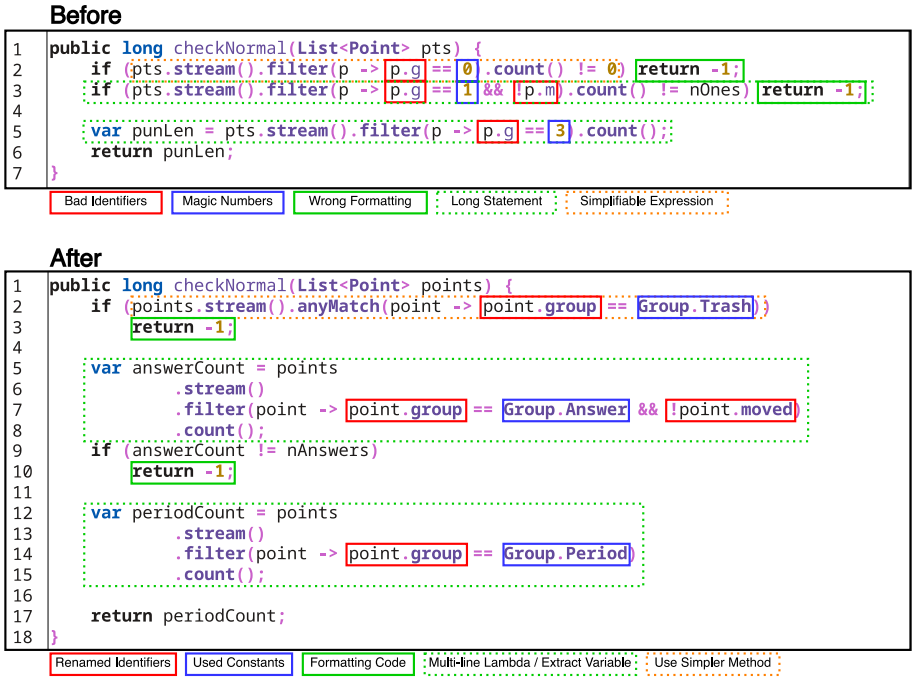

Fig. 1 Example of change aimed to improve readability

used in Vitale et al. (2023), while still considered a state-of-the-art model in 2023 (when the study was conducted), looks outdated in the modern LLMs landscape. For example, it has a very limited context window (512 tokens), which makes the approach hardly adoptable in practice for code-related tasks, since it can only "look" at very small inputs. This is also the reason why Vitale et al. trained it at "block-level granularity", meaning that a single diff representing a readability-improving commit was split into several training/test data points, each focusing on a specific code block of maximum 10 contiguous lines which were impacted by the change. This means that the model can not support a more complex readability-improving refactoring as the one presented in Fig. 1. In our work, we focused on a more realistic method-level granularity, in which we provide the LLMs with an entire method that has been subject to readability-improving changes in the commits' diff. While this may still look like limited in size, it is worth noting that readability-improving changes are usually well focused as compared, for example, to bug-fixes or architectural changes, which may involve even multiple files.

Despite employing newer LLMs being substantially larger than T5, *i.e.,* from $\sim$10$\times$ (0.5B) to $\sim$115$\times$ (7B) more trainable parameters (Hui et al. 2024), we only achieved up to EM=1.40%. Such a difference may be due to many differences in the experimental design, such as the choice of targeting more realistic (but also more challenging) method-level changes, rather than block-level ones. However, by investigating the root causes behind this major difference, we found out that in the original work (Vitale et al. 2023) the authors split the collected readability-improving commits randomly, with the possibility of having the commits from the same project both in the training and in the test set.

Recent work showed such a splitting could heavily bias the results (Shi et al. 2022a; Zhang et al. 2023; Zhu et al. 2024c). In the context of this work, this may be due to repetitive readability-improving commits performed within the history of certain projects, with some of them placed in the training and others in the test set. Indeed, our by-project splitting led to completely different (quantitative) findings.

Once fixed this issue, we moved to the main focus of our study, namely understanding whether fine-tuning an LLM for code readability improvement is still worthwhile having available general-purpose LLMs which, in theory, can automate several tasks in a zero-shot setting. We experiment in zero-shot the same LLMs we previously fine-tuned, namely those belonging to the Qwen2.5-Coder family (Hui et al. 2024) in four different sizes (0.5B, 1.5B, 3B, and 7B) as well as GPT-4.1, used via the OpenAI APIs. From a *quantitative* point of view (mainly EM predictions), we found that there is a slight advantage from fine-tuning models with code readability improvement examples, but not big enough to justify the costs. However, by manually inspecting some of the generated predictions, we quickly realized that quantitative metrics such as EM predictions tell very little about the quality of the readability-improving changes recommended by the LLMs. Indeed, in several cases the LLMs were recommending meaningful changes which, however, were different from those implemented by developers. Thus, we completely switched the focus of our study to a qualitative investigation, in which we inspected statistically significant samples of the predictions generated by different LLMs and different techniques (*e.g.,* fine-tuning *vs* zero-shot) to understand the extent to which the changes recommended by the LLMs do not change the behavior of the input code, as it would be expected from a readability-improving change. We found that all models change the behavior between 11.4% (Qwen2.5-Coder-7B instructed) and 16.2% (Qwen2.5-Coder-7B fine-tuned) of the cases, which clearly shows that developers should carefully check the models output before integrating the suggested changes.

Also, we performed a direct comparison between the best fine-tuned model we found, its zero-shot version, and the largest commercial LLMs considered in our work (GPT-4.1 in zero-shot) to qualitatively assess which of them was able to recommend the best readability-improving changes given a common set of code instances. The fine-tuned model achieved similar but slightly worse results as compared to the original developers, showing that it does learn their change characteristics. However, when directly comparing the fine-tuned and the zero-shot version of the same model, the latter achieved consistently better results, suggesting that fine-tuning, as proposed by Vitale et al., is less effective than zero-shot learning in this context. On the other hand, the two models in the zero-shot setting (Qwen2.5-Coder-7B instructed and GPT-4.1) achieve sensibly better results, not only when compared to the fine-tuned model, but also to the original developer, with GPT-4.1 emerging as the best model, at the moment, to improve code readability.

The main contributions of this paper are as follows:

- We differentially replicate and extend the study by Vitale et al. (2023) revisiting code readability improvement by building a larger dataset of readability-improving commits (spanning from 2015 to 2024) and introducing a method-level granularity setting with project-wise splitting to avoid data leakage.
- We provide an in-depth empirical evaluation state-of-the-art LLMs in code readability improvement. We considered both open-weight models (Qwen2.5-Coder, from 0.5B to

7B) in both fine-tuned and zero-shot settings, and a commercial closed-weight model (GPT-4.1). Our evaluation mainly focuses on human evaluation for assessing both behavior changes in the proposed improvements and the actual improvements in terms of readability.
– We release all data and code used in our study to foster future research on code readability improvement (Anonymous 2025).

2 Related Work

Software maintenance remains one of the most expensive activities in the software lifecycle. Erlikh (2000) showed that developers spend more time in maintenance activities than developing code. In fact, Oliveira et al. (2024) studied the importance of code understandability during code review, finding that many reviewers' comments are focused on the improvement of code understandability.

Given its importance, previous work aimed at defining approaches for automatically assessing code readability, essential to ease code understandability. Pioneering works, such as those from Buse and Weimer (2008, 2009), introduced the first approach for achieving such a goal through structural features (*e.g.,* line length and identifier length). Posnett et al. (2011) defined a simpler model for code readability by using three metrics: Halstead volume, entropy, and LOC. Later, Dorn (2012) achieved higher prediction accuracy proposing a new model leveraging visual, spatial, and linguistic features (*e.g.,* alignment of characters and indentation variation), and introduced a large dataset of manually-evaluated snippets in different programming languages. Scalabrino et al. (2016, 2018) defined a comprehensive model including all the state-of-the-art features, in which they also integrated a set of textual features (*e.g.,* consistency between comments and identifiers and comment readability). Mi et al. (2018) tested deep-learning techniques to predict code readability through Convolutional Neural Network (CNN) (Mi et al. 2018), visual, semantic and structural features (Mi et al. 2022), and graph neural networks (Mi et al. 2023). Recently, Sergeyuk et al. (2024a, 2024b) evaluated the alignment of existing code readability assessment models with developers, finding weak correlation between the latter and developers' ratings. They found moderate agreement between developers on key aspects that guide code readability assessments. Finally, Vitale et al. (2025a) studied personalized code readability assessments (*i.e.,* specialized for developer readability perspectives) through models tailored to developers previous assessments; they empirically show that also such models are not effective in automatically assessing code readability.

Vitale et al. (2023) defined the first and, to the best of our knowledge, only approach in the literature for automatically improving code readability. They collected a dataset of readability-improving commits and used it to fine-tune a T5 model. Since the publication of that work there have been significant advancements for the automation of code-related tasks using deep learning. Newer language models have been scaled both in the number of parameters (*i.e.,* billions vs millions) and the amount of training data (*i.e.,* billions vs trillions tokens). For example, the smallest variants of models such as CodeLlama (Roziere et al. 2023), StarCoder (Lozhkov et al. 2024), and DeepSeek-Coder (Guo et al. 2024a) contain up to $\sim 20\times$ more parameters than the pre-trained T5 adopted by Vitale et al., which only had 60M parameters. Additionally, these newer models are pre-trained on massive amounts of

text data including both natural language and code and are further fine-tuned for instruction-following to support conversational interactions. This allowed users to directly prompt the model to achieve a specific task *e.g.,* code generation, code summarization, and others.

The strong automation capabilities of these models used in a zero-shot setting has been also demonstrated for several software-related tasks. For example, Sun et al. (2024) showed that zero-shot prompting LLMs achieved strong effectiveness in code summarization according to human evaluation. Also, most of the proposed instruction-fine-tuned models showed strong performance on well-established benchmarks for both code generation and code editing (Guo et al. 2024a; Hui et al. 2024; Lozhkov et al. 2024; Roziere et al. 2023). Still, to the best of our knowledge, no previous work investigated the extent to which they can automatically improve code readability, thus understanding whether fine-tuning on available developer readability changes is actually needed for supporting this task. This is the aim of our work.

3 Study Design

The *goal* of our empirical investigation is to differentially replicate and extend the study by Vitale et al. (2023) considering the recent advances in LLM-based code automation. More specifically, we want to understand the extent to which (i) LLMs can actually recommend readability-improving changes; (ii) fine-tuning is actually needed and overcomes a zero-shot approach to the problem. In details, we aim at answering the following research questions (RQs):

RQ_1: *To what extent are LLMs fine-tuned and used in zero-shot prompting able to emulate the readability-improving changes made by developers?* This is the only RQ for which we completely rely on quantitative metrics (*e.g.,* EM and CrystalBLEU Eghbali and Pradel 2022) to (i) compare our findings to what reported by Vitale et al. (2023) in the fine-tuning scenario; and (ii) have a first (cheap) comparison between fine-tuning and zero-shot.

RQ_2 *To what extent do LLMs fine-tuned and used in zero-shot prompting change the code behavior?* By manually inspecting the code changes recommended by fine tuned LLMs and LLMs used in zero-shot, we assess whether the models try to improve readability or make other unrequested operations that change the code behavior.

RQ_3 *To what extent do LLMs fine-tuned and used in zero-shot prompting improve code readability as perceived by humans?* We qualitatively assess and compare the meaningfulness of the readability-improving changes (i) recommended by fine-tuned LLMs and those used in zero-shot setting, and (ii) implemented by human developers.

3.1 Context Selection

We introduce the context of our study in terms of used data and LLMs.

3.1.1 Training, Validation, and Test Sets

To collect the code readability improvement dataset, we rely on the methodology adopted by Vitale et al. (2023). In their work, the authors leveraged GitHub Archive (Grigorik 2012) to extract readability-improving commits spanning from January 2015 to December 2022.

The candidate commits were identified as those featuring keywords related to "readability" in the message (*e.g.,* readability, cleanup, formatting). Then, a two-stage filtering process was applied: (i) a word-blacklist was used to exclude false positives (*i.e.,* commits retrieved as readability-improvement which, however, relate to other types of changes); (ii) a NLP heuristic was employed to include only instances in which an *improvement* lemma or *code* lemma was linked to a *readability*-related one. Concerning the first point, we slightly improved it. In particular, by looking at several cases of false positives returned by the pipeline, we enlarged the sets of words in the blacklist, representing common terms often co-occurring with readability-related keywords but unrelated to the code readability. For example, we included terms related to: (i) GUI components (*i.e.,* banner, chart, div, svg, etc.) since sometimes developers use them in sentences line "resized svg icons for better readability"; and (ii) other quality aspects (*e.g.,* performance) to exclude tangled commits, which would be problematic both at test and training time (*e.g.,* "improved performance and refactored code to be more readable"). In total, we added 65 new terms in the blacklist (see replication package Anonymous 2025).

As per the NLP-based filter, we do not apply any change to the approach Vitale et al. defined. The basic idea behind their technique is to identify as relevant commits those in which a term related to "readability" relates to another term indicating improvement (*e.g.,* improved, increase, optimize) or to a term explicitly referring to a code element (*e.g.,* code, function, variable, statement). The full list of words are available in our replication package (Anonymous 2025). Specifically, each commit message is first divided in individual sentences. Then, for each sentence, a dependency tree is constructed using Python SpaCy. Finally, if the "readab-" node is linked via a dependency relation to either an improvement-related node or a code node the commit is retained, otherwise, it is discarded.

Subsequently, as in the original pipeline, we exclude commits (i) modifying too many files (>5, to reduce the chance of tangled commits), and (ii) modifying files not related to the programming language of interest (Java).

We also updated the dataset collected by Vitale et al. by including commits from January 2023 to December 2024, *i.e.,* we added two years of development on the whole GitHub. This resulted in additional ∼66k readability-improving commits (*i.e.,* +36%).

Vitale et al. also ran an experiment to validate the capabilities of their NLP-based technique for selecting readability-improving commits. Since we modified the procedure by including more keywords, we fully replicated it to assess its precision. Two authors manually analyzed a statistically-significant sample of 500 commits (95% confidence level, ±4.38% margin of error). The evaluators were asked to read the commit message, check the change diff, and report whether the *intention* of the developer was to improve readability (a boolean classification). The evaluators disagreed on 76 cases, which were carefully reviewed and discussed by both of them, finding an agreement. In total, the evaluators classified 452 commits out of 500 as readability-improvement (90.40%). Therefore, we can conclude that the percentage of actual readability-improving commits in the dataset we collected is between 86.02% and 94.78% (95% confidence level), which is slightly higher than the one reported in the original work (which was between 82% and 90.8%).

We build our dataset at method-level granularity, meaning that for a given readability-improving commit, we (i) identify all modified files, excluding added and deleted ones since we need the code both before (assumed lower readability) and after (assumed higher readability) the commit; (ii) find, within those files, the lines impacted by the changes; and

(iii) use *javaparser* (Javaparser n.d.) to map the impacted lines to the methods featured in the files. This gives use pairs of $\langle m_b, m_a \rangle$ with m_b representing a given method before the readability-improving changes and m_a its "improved" version. Remember that such a granularity is different compared to the one used in the replicated work (Vitale et al. 2023), which focused on block-level changes within the diff (*i.e.,* diff hunks) limited to up to 10 lines of code. In total, we collected 57,136 $\langle m_b, m_a \rangle$ pairs.

We then employ a data cleaning pipeline aimed at removing invalid instances. First, we excluded all pairs having an empty m_a or m_b. Indeed, while we only considered modified files in order to have both code before and after the readability-improving changes, it may still happen that within those files a method has been deleted (no m_a) or added (no m_b).

Then, we temporary abstract all strings, numbers, and dates in all collected methods <STRING>, <NUMBER>, and <DATE> tokens, respectively, and remove: (i) instances in which m_a is equal to m_b (*e.g.,* only a number has been changed); and (ii) duplicated pairs in the dataset, identified as pairs having the same m_a and/or m_b. For the latter, we only keep one instance in the dataset.

As done in recent work applying deep learning techniques to software-related tasks, we also apply additional filters to improve the data quality, removing pairs that (i) likely include self-admitted technical debt (SATD) (Mastropaolo et al. 2021; Shi et al. 2022b; Tufano et al. 2023) by simple string-matching with terms such as "`TODO`", "`FIX-ME`", etc.; (ii) contain non-ASCII characters (*e.g.,* emojis); (iii) present specific patterns (*e.g.,* "======="), frequently indicating toy examples (Tufano et al. 2023); and (iv) instances with commented-out code (Shi et al. 2022b). This left us with ∼45k $\langle m_b, m_a \rangle$ pairs.

We also remove instances composed by more than 1024 tokens (*i.e.,* as counted by the LLM tokenizer that we use). We choose 1024 as a limit since, during pre-processing, we found that ∼92% of the instances contained less than 1024 tokens. In the original work (Vitale et al. 2023) such a threshold was 512 which, in our dataset, would include only ∼ 69% of pairs. Finally, as done by Vitale et al., we exclude all trivial instances, namely those for which the token edit distance between m_b and m_a is less than 10.

Such a process left us with 31,781 $\langle m_b, m_a \rangle$ pairs, with m_b (*i.e.,* what will become the LLMs' input) containing up to 134 lines of code. To avoid data leakage, and differently from the previous work we are replicating (Vitale et al. 2023), we split the data into training, validation, and test sets by making sure that all pairs from a single repository are featured within one of these sets. We aimed at reaching the typical 80-10-10 proportion but, due to the repository-level splitting, we ended up with slightly different percentages (25,459 for training – 80.1%, 3,179 for validation – 10.0%, and 3,143 for test – 9.9%).

3.1.2 Large Language Models

Given the goal of comparing fine-tuned LLMs and LLMs used in a zero-shot setting, we started by selecting a set of open LLMs that can be used in both settings. We chose the Qwen2.5-Coder family (Hui et al. 2024), since it (i) features models in several different sizes; and (ii) achieved state-of-the-art performance across several well-established benchmarks (Hui et al. 2024).

Qwen2.5-Coder LLMs are built upon the Qwen2.5 architecture: Starting from Qwen2.5's model weights, these LLMs have been further pretrained on additional 5.5 trillions of tokens related to source code, text, and math data. In our experiments, we use the 0.5B, 1.5B,

3B, and 7B trainable parameters variants. This also allows to observe how the model size impacts the quality of the readability improvements with and without fine-tuning.

We use each model both (i) in zero-shot prompting, and (ii) after fine-tuning it with the training set of 25,459 $\langle\ m_b, m_a\ \rangle$ pairs previously described. We fine-tune each model for 5 epochs using a batch size of 16 and a the maximum length set to 1024 tokens. Following previous work (Lin et al. 2024; Kavathekar et al. 2025; Ren et al. 2025; Silva et al. 2025; Wang et al. 2025a, b, c), we use AdamW (Loshchilov and Hutter 2017) as optimizer and set 2e-5 as learning rate. Additionally, we set a warmup ratio of 10% and a cosine learning rate scheduler. We use BF16 precision to accelerate the training process. We employ early stopping to prevent overfitting (Prechelt 2002). We assess the effectiveness of the model after each training epoch by calculating the number of correct predictions achieved on the validation set. A correct prediction corresponds to the model's ability to implement readability improvements as the developer did (exact match).

The fine-tuning procedure is stopped and the best-performing checkpoint is selected if there is no gain in correct predictions after three epochs.

We also include in our study GPT-4.1 (OpenAI n.d.) as a representative of closed-weights and general-purpose models. GPT-4.1 was the flagship OpenAI model when the experiment has been ran. We use GPT-4.1 only in zero-shot setting.

Table 1 reports the models included in our study, with the name we will use in the remainder of the paper to refer to them.

3.2 Data Collection & Analysis

All non-fine-tuned models (*i.e.,* the four Qwen2.5-Coder variants and GPT-4.1) used in zero-shot have been instructed with the following prompt: "*Improve the readability of the following Java code and return ONLY the updated method:* `{method}`", where `{method}` is the m_b in the collected pairs (*i.e.,* the method in need of readability improvements). In contrast, for the fine-tuned models, as set during the training, we simply provide them with m_b as input using the prompt: "*Improve the readability of the following Java code:* `{method}`". We defined the zero-shot prompt by iterative refinement: The specification *"return only the updated method"* was necessary since, without it, the model tended to return multiple methods as an output. This was not the case for the fine-tuned models, that have been explicitly trained to generate a single method m_a given m_b, thus not requiring in the prompt a clarification about the expected output format. We use greedy-decoding to generate the predictions (temperature = 0) to foster replicability.

Table 1 Models considered in our experiments

Name	Family	Size	Modality
Q_{ft}-0.5B	Qwen2.5-Coder	0.5B	Fine-Tuned
Q_{ft}-1.5B	Qwen2.5-Coder	1.5B	Fine-Tuned
Q_{ft}-3B	Qwen2.5-Coder	3.0B	Fine-Tuned
Q_{ft}-7B	Qwen2.5-Coder	7.0B	Fine-Tuned
Q_{in}-0.5B	Qwen2.5-Coder	0.5B	Zero-shot
Q_{in}-1.5B	Qwen2.5-Coder	1.5B	Zero-shot
Q_{in}-3B	Qwen2.5-Coder	3.0B	Zero-shot
Q_{in}-7B	Qwen2.5-Coder	7.0B	Zero-shot
GPT-4.1	GPT-4.1	N.A	Zero-shot

3.2.1 RQ_1: Reproducing Developers' Changes

To answer RQ_1, we compute for all predictions generated by each model on our test set (i) the percentage of exact matches, and (ii) descriptive statistics about the CrystalBLEU score (Eghbali and Pradel 2022). The former compares the candidate improvement recommended by LLM with the changes implemented by developers in the mined commit, considering the candidate correct only if it is identical to the developer's change. This is a quite strict metric, representing a lower bound for the performance of the LLMs. CrystalBLEU, instead, is a similarity metric between the prediction (candidate) and the expected output (reference). Differently from BLEU (Papineni et al. 2002), which primarily focuses on matching n-grams, CrystalBLEU mitigates the influence of trivially shared n-grams *i.e.,* common sequences that occur frequently among source code artifacts due to syntactic sugar or programming languages, but that do not necessarily indicate semantic similarity.

3.2.2 RQ_2: Changing Behavior

To answer RQ_2, in line with previous work (Vitale et al. 2023), we rely on manual inspection of the generated predictions with the goal of assessing how frequently the LLMs generate code changes which are certainly wrong, since they alter the behavior of m_b, something which should not happen with a readability-improving change. We opted for manual inspection instead of automated testing (*e.g.,* unit test execution) because the latter would have introduced several challenges. First, we would have needed to have unit tests targeting the method within the original repository, a requirement that is not always satisfied in open source software, especially in less mature projects. Second, the tests available should be adequate enough (*e.g.,* they should have high branch coverage), which literature shows is often not true in practice (Wang et al. 2024). Besides, test adequacy would still be insufficient to guarantee that tests would allow to find bugs in the improved version of the code. Having considered all those aspects, we preferred a double and independent manual inspection, which – although requiring more effort – is more accurate than testing in finding bugs and allows for more fine-grained checks (*i.e.,* identifying the precise nature of the behavioral change) in this context.

Running manual analysis on a sufficiently big sample of predictions for all studied models would have been too expensive. Thus, based on the idea that larger models tend to perform better in automating code-related tasks (Chen et al. 2021), we focus on the manual inspection of Q_{ft}-7B, Q_{in}-7B, and GPT-4.1. Basically, we manually inspect a sample of readability-improving changes recommended by the largest open model used in our study (7B) in both its versions (*i.e.,* fine-tuned and used in zero-shot), as well as by the only commercial model we adopt (GPT-4.1).

To sample the instances to inspect, we started by selecting 500 $\langle\, m_b, m_a \,\rangle$ pairs in which we could confirm that the developer-introduced changes did not alter the code behavior. This was a way to consider cases in which we know that the LLM at least had the chance to implement meaningful changes without impacting the code behavior. Indeed, note that this is not guaranteed for all pairs we collected, since, as we documented, errors are made by the automated collection pipeline. The selection was made by two authors, who independently inspected 500 instances as a starting point with the goal of labeling them as behavior-preserving or not. To ensure that they had the same understanding of the task, the evaluators

had a meeting where a series of examples were provided and discussed, clarifying the criteria to define whether a candidate improvement changes the behavior. The evaluators agreed on the following process. First, they examined the instruction blocks in m_b and looked for their corresponding parts in m_a, and vice versa. If any block in one version did not have a clear counterpart in the other version, they concluded that the behavior had changed. Otherwise, they examined the equivalence of the corresponding blocks (*e.g.,* they checked that the actual parameters of method calls were the same). If any difference was observed, they concluded that the behavior had changed. Finally, if no relevant differences were identified, they concluded that the behavior was preserved. Take as an example a method that includes a check on the presence of a column of a CSV file before returning its mean, and a candidate that removes such a check. In this case, the evaluators can confirm that the candidate changes the behavior logic of the original code because there is no mapping for that instruction block (the `if` condition). Cases of disagreements (15) have been resolved via open discussion. Then, the 121 pairs labeled as "changing the behavior" have been replaced by new ones, iterating the process until the 500 target was reached. In total, this required the inspection of 672 pairs and the solving of 17 cases of disagreement.

Once collected, the 500 $\langle\ m_b, m_a\ \rangle$ pairs, providing a statistically significant sample (95% confidence, ±4% margin of error), based on the previously defined criteria, were independently inspected by two authors for all predictions generated by the three models, for a total of 1500 predictions each. The goal of the manual inspection was the same of the one performed for the human-implemented changes: Labeling each prediction as behavior-preserving (thus, possibly a correct readability-improving change) or not (certainly wrong). Again, 26 cases of disagreement were resolved through an open discussion.

We report, for each model, the percentage of cases in which their predictions were behavior-changing (the lower the better). In addition, we conduct a Fisher exact test (Fisher et al. 1956) as an exploratory check to observe the general trends in behavior-preserving differences between models (three comparisons: Q_{ft}-7B vs. Q_{in}-7B, Q_{ft}-7B vs. GPT-4.1, and Q_{in}-7B vs. GPT-4.1). We adjust *p*-values using the Benjamini-Hochberg procedure (Benjamini and Hochberg 1995) to account for multiple comparisons. We reject the null hypothesis (*i.e.,* there is no difference) if the *p*-value is lower than 5%.

3.2.3 RQ_3: Readability Improvement Capabilities

Finally, to answer RQ_3 we run again a qualitative analysis of the candidate improvements generated by the same three models subject of RQ_2 (Q_{ft}-7B, Q_{in}-7B, GPT-4.1) starting from the same subset of 500 pairs for which humans did not change the code behavior. However, RQ_3 aims at (i) directly comparing the three models among them, to see which ones provides the best readability-improving changes, (ii) compare the readability improvements achieved by models with those achieved by humans, and (iii) quantify the effect of fine-tuning comparing the fine-tuned and zero-shot versions of the same model (Q_{ft}-7B and Q_{in}-7B). Thus, we only selected the subset of 500 pairs for which all three models generated a behavior-preserving code change. This resulted in 342 instances, that additionally assures a statistically representative sample with a 95% confidence level and a 5% margin of error. In this way, we can focus on the eventual readability improvement provided by the models. Basically, the combination of RQ_2 and RQ_3 will tell us which models generate better readability-improving changes (RQ_3), how do they compare with the human-written

improvements (RQ_3), how fine-tuning influences model effectiveness (RQ_3), and the extent to which they avoid the side effect of changing code behavior (RQ_2).

We involved three authors as annotators, who independently performed the following task. For each $\langle\ m_b,\ m_a\ \rangle$ pair, they inspected m_b (*i.e.,* the code before the readability-improving changes), m_a (*i.e.,* the code implementing the human-written improvements), and the three improvements recommended by Q_{ft}-7B, Q_{in}-7B, and GPT-4.1. The annotators assigned a score on a scale between -4 to 4 to each of the four readability-improved candidates. The guidelines to score each candidate were as follows:

- **0**: Used to indicate no improvement in readability as compared to m_b (*e.g.,* no changes were implemented).
- **1–4**: Used to indicate an improvement in readability as compared to m_b. The higher the score the stronger the magnitude of the perceived improvement. Having four values allowed participants to define a sort of ranking among the four candidates in the case all of them increased readability but with different magnitudes.
- **–1 to –4**: Used to indicate a deterioration in readability as compared to m_b. The lower the score the stronger the magnitude of the perceived deterioration.

Participants were allowed to give the same score to two solutions if they perceived them as equally good (or bad). The solutions to score were shown in random positions, not allowing the annotators to know the "generator" of the solution (either a human or one of the three LLMs). Similarly to the previous manual analysis, the evaluators had a meeting in which the aim and the procedures of the task were explained and agreed. To make this, a series of examples were shown, everyone manifested their code readability preferences and were guided toward the compilation of the scores. Once all the evaluators felt confident as for the process, the manual analysis proceeded.

For each evaluator and instance, given the scores assigned, we extracted the preference ranking. For example, given the scores $c_1 = 4$, $c_2 = 2$, $c_3 = 1$, $c_4 = 2$, the ranking is $c_1 > c_4 = c_2 > c_3$. We do not compute agreement per instance, since readability is inherently subjective (Sergeyuk et al. 2024a; Vitale et al. 2025a). Instead, we treat each evaluator ranking for each instance as an independent preference. Thus, for *n* instances annotated by *m* evaluators, we collect a total of $n \times m$ rankings, which we then aggregate across the entire set of samples.

This allows us to apply Condorcet-based voting rules (Krishnamoorthy and Raghavachari 2005) to aggregate the rankings. This is done by constructing a matrix in which each cell (i, j) represents the number of times candidate c_i is preferred over c_j across all evaluator rankings. We first check for a *Condorcet winner* (Elkind et al. 2015), namely a candidate that beats all others in head-to-head comparisons. If such a winner exists, it is ranked first. Otherwise, we use the *Copeland rule* (Saari and Merlin 1996), which assigns each candidate a score equal to the number of pairwise wins minus losses. The candidate with the highest Copeland score is ranked first, followed by others in descending order. This final ranking provides a global representation of how each model-generated and human-implemented changes are perceived in terms of readability improvements. We report the ranking of the models, as well as the Condorcet matrix. Note that this also naturally produces pairwise comparisons among any two candidates. This allows us to directly assess, for instance, how the developer changes compare to those generated by GPT-4.1, or how Q_{ft}-7B performs to

its counterpart Q_{in}-7B. The latter comparison is particularly relevant, as it directly quantifies the effect of fine-tuning, one of the objectives of RQ_3.

4 Results

In this section we present the results of our study, answering the research questions defined in Section 3.

4.1 RQ_1: Reproducing Developers' Changes

We report in Table 2 models' performance as assessed by the previously described metrics, *i.e.,* Exact Match (EM) and CrystalBLEU.

First, it is clear that all models are not effective at reproducing the exact readability-improvement changes implemented by developers. Indeed, no model can achieve an EM score higher than 1.40%. This means that the best model (*i.e.,* Q_{ft}-7B) perfectly emulates code readability improvements made by developers for 44 instances out of the ∼3k featured in our test set. As per the CrystalBLEU values, while they look pretty high (*e.g.,* CB = ∼ 74.3 for Q_{in}-0.5B), it must be considered that the LLMs received m_b as input (*i.e.,* the method before the readability improvement) and have been instructed (or fine-tuned) to implement readability-improving changes, not expected to change m_b too much. Indeed, the commits we mined focused on readability-improving changes as well, thus resulting in an m_a (the method after the readability improvement) relatively similar to m_b. The m_a method represents our "oracle" to assess CrystalBLEU, and it is expected by construction to have some form of similarity with the LLMs' prediction, simply due to the common starting point (m_b). Thus, while the CrystalBLEU values may provide hints about which models perform the best, they should not be used to assess LLMs' performance in absolute terms.

When comparing fine-tuned to instructed models, we can observe that the former achieve higher performance than the latter. The smaller variant of Qwen2.5-Coder (0.5B) achieves ∼46% higher exact match when it is fine-tuned as compared to its zero-shot counterpart. This is even more clear for the biggest variant (7B), for which the EM goes from 0.06% up to 1.40%. When considering EM values, GPT-4.1 which is worse than all the fine-tuned models. As we will see, this quantitative assessment, central in many papers on DL-based automation of code-related tasks (Guo et al. 2024b; Li et al. 2022; Prenner and Robbes 2024; Zhu et al. 2024b), only says a little part of the story.

Table 2 RQ_1. Exact match and crystal BLEU achieved by the experimented models

Mode	Model	Exact Match	Crystal BLEU
Zero-shot	Q_{in}-0.5B	0.70%	74.3%
	Q_{in}-1.5B	0.10%	62.2%
	Q_{in}-3B	0.13%	51.1%
	Q_{in}-7B	0.06%	55.6%
	GPT-4.1	0.13%	54.7%
Fine-tuned	Q_{ft}-0.5B	1.02%	71.1%
	Q_{ft}-1.5B	0.64%	67.4%
	Q_{ft}-3B	1.37%	70.7%
	Q_{ft}-7B	1.40%	69.5%

When focusing on fine-tuned models, we observed that the bigger the models, the higher their accuracy in reproducing the developers' changes. The only exception to this trend is the 1.5B model, which achieves worse results than Q_{ft}-0.5B.

The biggest Qwen2.5-Coder variant (7B) achieves the highest EM score. Note, however, that the CrystalBLEU of all fine-tuned models is comparable, with the smallest model (Q_{ft}-0.5B) achieving the highest score. When analyzing the predictions, we noticed that smaller fine-tuned models tend to make less changes than bigger models, or no changes at all. For example, Q_{ft}-0.5B does not change the input code in ∼20% of the instances, while Q_{ft}-7B does the same only in ∼5% of the cases.

For the reasons previously explained (*i.e.,* the high similarity between m_b and m_a), this actually helps Q_{ft}-0.5B in terms of achieved CrystalBLEU.

As for the instructed models, the results show that the smallest model *i.e.,* Q_{in}-0.5B reproduces the readability improvements made by real developers more often than its bigger variants and even than GPT-4.1. The same is true for CrystalBLEU. This is again due to the lower number of changes implemented by the smaller LLMs. Indeed, we noticed that developers modified ∼20% to ∼40% of the input lines (*i.e.,* m_b), while instructed models tend to provide more significant changes, usually modifying ∼80% of the input, or even revisiting the whole method. So, they get a lower CrystalBLEU and low chances of exact match.

The fine-tuned models, having been instructed on the developers' commits, tend instead to implement a number of changes aligned with those of the developers (and, thus, the higher EM and CrystalBLEU).

A last point of discussion is the comparison of the results we achieved in terms of EM with those reported in the work by Vitale et al. (2023). In their work, using a smaller model and a comparable amount of fine-tuning data (24,945 training instances in Vitale et al. (2023) *vs* 25,459 in our work), the authors reported an EM score of ∼20%, while our best model, being 117 times bigger, achieved 1.40%. Our assumption is that their choice of randomly splitting the readability-improving commits between training and test (rather than by project, as we do), substantially boosted the performance due to repetitive changes implemented by developers over the history of the same software system. To better investigate such a conjecture, we tried to randomly split our dataset, and replicated our assessment with the best fine-tuned model we in our study (*i.e.,* Q_{ft}-7B). We achieved EM=14.4%, confirming that the choice of the splitting strategy significantly affects model performance. Note that our EM is still lower than the one obtained by Vitale et al. (2023), probably due to the most complex experimental setting we adopt, *i.e.,* we experiment with readability-improving changes affecting entire methods (up to > 100 lines of code), while the previous work focused on code chunks of up to 10 lines (Vitale et al. 2023). Note that we can not conclude with certainty that this is the case because we did not study the impact of each modification we made in the design as compared to the previous work we are differentially replicating since we do not want to test whether this design is better than the other – it is for theoretical reasons, not for empirical ones.

In short, the findings or RQ_1 provided with more questions than answers, showing the unsuitability of quantitative metrics to assess the effectiveness of techniques for the automated improvement of code readability. The following RQs, having a strong qualitative focus, will help in better understanding the quality of the LLMs' recommendations.

Answer to RQ$_1$. While fine-tuning helps LLMs to better imitate the readability-improving changes implemented by developers, the low percentage of EM achieved makes it difficult to make any claim about the effectiveness of the techniques. Indeed, as we observed, the amount of changes that different approaches implement in the inputted code substantially differs for instructed models, which may still be effective in improving readability despite moving away from the changes implemented by developers.

4.2 RQ$_2$: Changing Behavior

Table 3 shows the percentage of cases in which a change implemented by the three LLMs considered in RQ$_2$ (*i.e.,* Q_{ft}-7B, Q_{in}-7B, and GPT-4.1) result in a change of behavior in the impacted code, something not expected from a readability-improving change. Table 3 also provides the results of the Fisher exact test, with the arrows indicating more (↑) or less (↓) cases in which the LLM on the row impacted the code behavior as compared to the LLM on the column. For example, when comparing Q_{ft}-7B *vs* Q_{in}-7B, the former produces more changes in behavior than the latter (↑), but such a difference is not statistically significant (p-value = 0.10).

The model that more rarely modifies the original behavior of the code is Q_{in}-7B (11.4% of the instances), followed by GPT-4.1 (13.2%), and Q_{ft}-7B (16.2%). This is a first finding suggesting that the fine-tuned models may actually perform worse than the instructed one. This may be due to several reasons, such as the attempt to replicate complex changes seen during fine-tuning (*e.g.,* extract variable refactorings which may be difficult to implement without introducing errors) or to noise in the fine-tuning dataset that, as said, is far from perfect despite the improvements we implemented in the data collection pipeline. However, this basically holds for all kind of automatically-collected training datasets (Li et al. 2024; Penedo et al. 2024).

Also, it is also worth noting that none of the statistical tests we run highlight a significant difference in the likelihood of observing code behavioral changes implemented by the three different LLMs. Indeed, the only significant p-value we obtained was a 0.035 when comparing Q_{ft}-7B *vs* Q_{in}-7B. However, after the adjustment due to the multiple comparisons, it became 0.10. Increasing the sample size might result in a significant difference. However, note that this would not change the main finding: All models change behavior in a non-negligible fraction of cases (>10% of the cases), with Q_{ft}-7B tending to introduce behavioral changes more frequently.

Table 3 RQ$_2$. How frequently each model changed the behavior of the code and the adjusted p-values of the comparisons between pairs of models (↑and ↓indicate more and less behavior-changing instances, respectively)

Model	Different	Comparison (p-values)		
	Behavior	Q_{ft}-7B	Q_{in}-7B	GPT-4.1
Q_{ft}-7B	16.2%	-	↑	↑
Q_{in}-7B	11.4%	↓	-	↓
GPT-4.1	13.2%	↓	↑	-

A concrete example of change implemented by Q_{ft}-7B that impacts the code behavior is reported in Listing 1. This is only a small part of the actual diff, in which some readability-related changes were also implemented. However, the modification of the `return` statement clearly affects the code behavior, and has no evident link to any readability-improving attempt.

Listing 1 Q_{ft}-7B change impacting code behavior

```
- return asString.toString();
+ return asString.toString().replaceFirst("\n", "");
```

In general, there is a clear outcome that can be derived from our analysis: No model, whether fine-tuned or used in a zero-shot setting, is capable of improving code readability without inadvertently changing the underlying behavior in some cases.

Thus, LLMs for readability improvement are definitely not trustworthy and require developers double-checking the implemented changes.

Answer to RQ$_2$. LLMs modify code behavior in ∼11% to ∼16% cases when attempting to improve readability. No major changes are observed between fine-tuned models and models used in zero-shot.

4.3 RQ$_3$: Readability Improvement Capabilities

Table 4 reports the percentage of cases in which readability has been increased, decreased, or kept unchanged after the changes implemented by the developer (Dev.) and each analyzed model (Q_{ft}-7B, Q_{in}-7B, GPT-4.1) according to the manual inspection performed by three evaluators (E1, E2, and E3).

A first general observation is that E1 perceived readability improvements as neutral in a minority of the cases. As a result, E1 more often reported increased/decreased readability. Except for this difference, the three evaluators tend to agree that both changes implemented by the LLMs and by the developers tend to increase readability more often than they

Table 4 RQ$_3$. Percentage of cases in which developers and LLMs implemented changes that decrease, do not change, or increase code readability.

	Dev.			Q_{ft}-7B		
	Decreased	Same	Increased	Decreased	Same	Increased
E1	13%	10%	77%	19%	20%	61%
E2	5%	64%	31%	3%	70%	27%
E3	5%	64%	31%	3%	71%	26%
	Q_{in}-7B			GPT-4.1		
	Decreased	Same	Increased	Decreased	Same	Increased
E1	12%	4%	84%	11%	2%	87%
E2	4%	39%	57%	2%	29%	69%
E3	8%	38%	54%	1%	32%	67%

decrease it. While this is to be expected for the developer, it is was not granted for the three models we experimented with.

Another clear trend is that the changes made by the fine-tuned model (Q_{ft}-7B) appear to have analogous characteristics to the changes made by the developer: According to E2 and E3, most suggested changes are not affecting readability, while most of the reminder increase it. The fine-tuned model, however, acts as a "weak" version of the developers, since all evaluators agree that it performs worse.

All evaluators agree, instead, that the instructed models (Q_{in}-7B and GPT-4.1) increase readability more often than the both the fine-tuned models and the developers. GPT-4.1 exhibits the most consistent perception of improvement across all evaluators (between 67% and 87%). This suggests that LLMs do have a notion of what it means to improve code readability even though they are not explicitly trained to do so, and such a notion might be more objective (thus more generally accepted), than what readability represents for developers.

In Fig. 2 we show the comparisons between all pairs of contrasted changes (win-tie-lose), as evaluated by each evaluator and by averaging their preferences. In particular, each block of four bars represent the comparison between two LLMs or between the developers-implemented changes and those suggested by an LLM. For example, the top block compares the developers' changes with those recommended by Q_{ft}-7B. The four bars represent (from top to bottom) the assessments given by the three evaluators (E1, E2, E3) and their average. Finally, the color schema and percentages represent the cases in which the readability-improving changes recommended by the candidate on the left (Dev. in our example) is considered better than (green), equivalent to (yellow), or worse than (red) the candidate on the right (Q_{ft}-7B).

When comparing Dev. and Q_{ft}-7B, the developer is slightly better than the fine-tuned model. All evaluators agree that the two have similar characteristics, aligning with our earlier observation that the fine-tuned model behaves as a weaker approximation of developers changes.

> ⚖ **Finding 1.** Developers changes are preferred to those by Q_{ft}-7B.

When comparing Dev. vs Q_{in}-7B and Dev. vs GPT-4.1, the instructed models clearly outperform developers changes, according to all evaluators. In particular, GPT-4.1 is strongly preferred (64% wins, 22% ties, 14% losses), while Q_{in}-7B also consistently outperforms developer changes, albeit with smaller margins. This show that instructed models can provide readability improvements that are often perceived as superior to developer ones. Figure 3 shows a concrete example of readability-improving changes introduced by the developer (top) and GPT-4.1 (bottom). The developer focused on reformatting the `if` condition to better highlight, via parenthesis, the precedence that the && operator has in Java over the || operator. For this reason, the diff does not alter the method's behavior, but it makes more explicit the order in which the `if` condition is evaluated. GPT-4.1 instead, extracts the evaluated (sub-)conditions into descriptive boolean variables thus avoiding unnecessary nesting and improving the code understandability. All three evaluators were in agreement, preferring the GPT-4.1's solution.

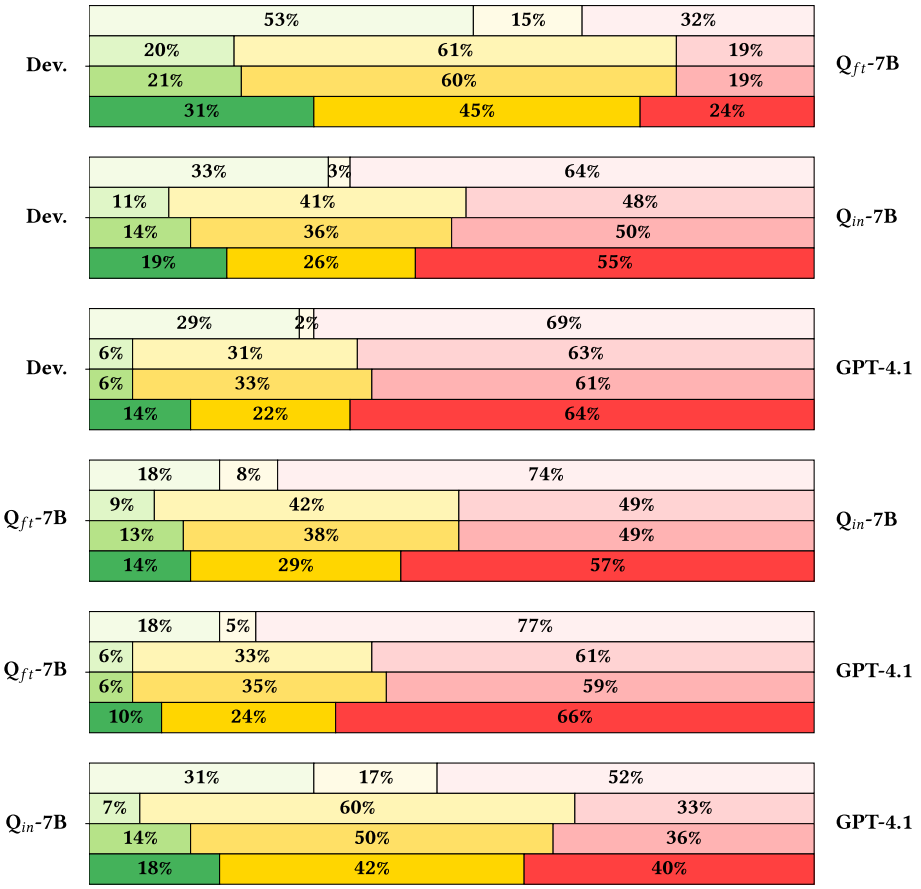

Fig. 2 Comparison between pairs of models. Each group of bars reports the percentage of cases in which the candidate on the left is perceived better than (green), equivalent to (yellow), or worse than (red) the candidate on the right according to E1, E2, and E3 – the last bar contains the average

> ⚖ **Finding 2.** Instructed models changes are preferred to those by Dev.

When comparing Q_{in}-7B and GPT-4.1, the two instructed models exhibit similar behavior, with GPT-4.1 generally preferred by all evaluators (40% wins, 42% ties, 18% losses). The high percentage of ties suggests that, despite being probably smaller in parameter size, Q_{in}-7B can generate readability improvements comparable to those of GPT-4.1, confirming its strong zero-shot capabilities.

> ⚖ **Finding 3.** GPT-4.1 changes are preferred to those by Q_{in}-7B.

A particularly relevant comparison is the one between the two versions of the same model, Q_{ft}-7B and Q_{in}-7B. Despite sharing the same architecture and pre-training, their readability improvement capabilities change noticeably once fine-tuning is performed.

Fig. 3 Example of readability-improving changes introduced by the developer (top) and by GPT-4.1 (bottom)

Across all evaluators, the zero-shot version (Q_{in}-7B) is consistently preferred, indicating that fine-tuning, as performed in our work, does not enhance, and may even hinder the model's ability to produce effective readability-improving changes. This suggests that fine-tuning may have led the model to over-specialize on developer-introduced changes styles, patterns, and quality, thereby limiting its ability to generalize and to produce broader and effective readability improvements.

> ⚖ **Finding 4.** Q_{in}-7B changes are preferred to those by Q_{ft}-7B.

Finally, Fig. 4 depicts the pairwise matrices for all candidates, separately for each evaluator and for the combined assessment (from left to right). Looking at the results from E1, GPT-4.1 achieves the highest number of wins across all other candidates. Q_{in}-7B also obtains an high number of wins, particularly against Dev. and Q_{ft}-7B, while the developer and fine-tuned model obtain fewer wins. GPT-4.1 and Q_{in}-7B are the most successful candidates also for E2 and E3. However, differently from E1, Dev. and Q_{ft}-7B achieve fewer wins in head-to-head comparisons. The rightmost matrix aggregates the counts of all evaluators, providing a representation of the overall preferences. This matrix confirms that GPT-4.1 reports the highest pairwise win counts against every other candidate (657 against the developer, 675 against Q_{ft}-7B, and 414 against Q_{in}-7B). Similarly, Q_{in}-7B reports a

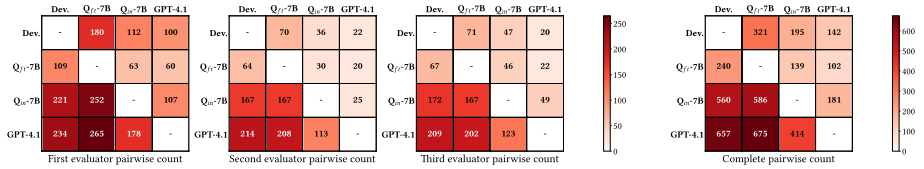

Fig. 4 Number of times the approach on the row is considered better than the approach on the column according to each evaluator (matrices for E1, E2, E3, respectively, from the left). The rightmost matrix aggregates (sums) the other matrices

high number of wins, mostly against the other two candidates, *i.e.,* the developer (560) and Q_{ft}-7B (586).

Given such a result, GPT-4.1 emerges as the Condorcet winner, since it is preferred over all other candidates in head-to-head comparisons. Finally, applying the Copeland rule to the aggregated preferences, the global ranking of the four candidates is the following, which closely reflects what we previously discussed: GPT-4.1 is ranked first, followed by Q_{in}-7B, developers, and Q_{ft}-7B.

Answer to RQ$_3$. LLMs can effectively improve code readability as perceived by humans, with instructed models being better than fine-tuned ones. GPT-4.1 achieves the best overall results, also beating the developers-implemented changes.

5 Discussion

The main outcome of our study is that, in the current LLMs landscape, fine-tuning following Vitale et al. approach (Vitale et al. 2023), namely training on readability-improving commits, is not really needed to teach LLMs how to improve code readability. Instructed and ready-available LLMs are more effective in this task. Although this may appear counterintuitive, a closer look at the results shows that the main issue here are the improvements introduced by the original developers from which the fine-tuned model learns the readability-improvement patterns.

Table 5 reports the statistics related to the instances analyzed in RQ$_3$, those of the actual fine-tuning data. On one hand, we observe that developer changes, both in the training data and in part of the test data (Train and Dev. rows), tend to be small, with low mean edit distances and few modified lines, as expected from commit-derived instances. Since the fine-tuned model Q_{ft}-7B learns directly from this data, it inherits these same characteristics, generating similarly minor and localized edits (Q_{ft}-7B row). On the other hand, instructed models (Q_{in}-7B and GPT-4.1) demonstrate different characteristics, generating larger changes. Human evaluators consistently preferred instructed models changes, implying that they perceive larger modifications as more effective readability improvements. Consequently, the low effectiveness of fine-tuning is not due to the process itself but to the limited nature of the data on which it relies. In short, fine-tuning is not the issue; available fine-tuning data is. Fine-tuning teaches the model to replicate the small changes that developers make in practice, rather than larger readability improvements that humans prefer.

Table 5 Mean edit distance (ED) and mean line-level change statistics for developer- and model-generated readability improvements. *Removed*, *added*, and *updated* refer to distinct line-level operations, while *Modified* indicates their combined total. Results refer to the instances analyzed in RQ_3, while *Train* row summarizes the trainin data

Source	ED	Line-level operations			Total
		⊖ Removed	⊕ Added	Δ Updated	± Modified
Train	109.29	0.26	3.93	0.40	4.59
Dev.	96.88	0.14	3.94	0.22	4.30
Q_{ft}-7B	63.08	0.14	3.16	0.28	3.58
Q_{in}-7B	264.57	0.58	15.23	1.39	17.20
GPT-4.1	268.72	0.47	16.35	0.51	17.33

Future research should refine fine-tuning based on larger and higher-quality datasets, or with human-aligned supervision strategies, teaching models not what humans do, but what they prefer.

Thus, we conclude that practitioners should at least try to use ready-available LLMs in a zero-shot setting since they are, most likely, more than enough to help them improving readability.

Our findings also show that open models (*i.e.,* Qwen2.5-Coder) are not much less effective than closed models (*i.e.,* GPT-4.1), and still provide valuable recommendations. Anyway, practitioners should always carefully inspect the LLMs' recommendations since, on average, once out of ten times they are adopted their modifications change the code behavior, possiblty introducing bugs.

As it is well-known in the literature (see *e.g.,* Sergeyuk et al. 2024a; Vitale et al. 2025a), readability is an inherently subjective aspect of the source code. This is reflected from our results of RQ_3: Two evaluators were closely aligned (the Cohen's k between E2 and E3 is $\sim$ 0.72), while E1 was not (Cohen's k is $\sim$0 with both E2 and E3, respectively). Indeed, while all evaluators agreed on the overall results (*i.e.,* the final candidates' ranking), there were major differences in the readability assessment of E1. Let us consider the example in Fig. 5. First, it can be noticed that the output of the fine-tuned model resembles the one made by the original developer more than the LLMs used in zero-shot. Second, while E1 prefers the output of the fine-tuned model, E2 and E3 prefer the code recommended by the same model (*i.e.,* Qwen2.5-Coder) used in zero-shot. In this case, E1 prefers simpler indentation changes, while E2 and E3 prefer a more structured refactoring also involving renamings and the addition of comments. Thus, it is still possible that some practitioners might find the readability improvement suggestions provided by a zero-shot LLM not enough and might prefer solutions tailored on their preferences or on the preferences of a specific community. Fine-tuning a model on data collected from a specific repository might help to achieve this. This is something we plan to investigate as part of our future work.

Another outcome of our study is related to the metrics adopted for evaluating code readability improvement models. As we expected and also noted in previous work (Vitale et al. 2023), both exact match and similarity-based metrics (CrystalBLEU, in our case) are not good metrics for estimating the quality of a model for automated code readability improvement. Our results even show that those metrics are almost inversely related to the actual capabilities of the model observed in our qualitative analysis (RQ_3). GPT-4.1 has almost no exact matches (only 0.13%), while the best fine-tuned model (*i.e.,* Q_{ft}-7B) appears to be much better based on such a metric.

Code to Improve

```
private String cleanWord(String next) {
    // First, if there is an apostrophe, the substring
    // before the apostrophe is used and the rest is ignored.
    // Words are converted to all lowercase.
    // All other punctuation and numbers are skipped.
    String ret;
    int inx = next.indexOf('\'');
    if (inx != -1) ret =
        next
          .substring(0, inx)
          .toLowerCase()
          .trim()
          .replaceAll("[^a-z]", ""); else ret =
        next.toLowerCase().trim().replaceAll("[^a-z]", "");
    return ret;
}
```

Dev.

```
private String cleanWord(String next) {
    String ret;
    int inx = next.indexOf('\'');
    if (inx != -1) ret =
        next
           .substring(0, inx)
           .toLowerCase()
           .trim()
           .replaceAll("[^a-z]", "");
    else ret =
        next.toLowerCase().trim().replaceAll("[^a-z]", "");
    return ret;
}
```

Q_{ft}-7B

```
private String cleanWord(String next) {
    // First, if there is an apostrophe, the substring
    // before the apostrophe is used and the rest is ignored.
    // Words are converted to all lowercase.
    // All other punctuation and numbers are skipped.
    String ret;
    int inx = next.indexOf('\'');
    if (inx != -1) {
        ret = next.substring(0, inx).toLowerCase().trim().replaceAll("[^a-z]", "");
    } else {
        ret = next.toLowerCase().trim().replaceAll("[^a-z]", "");
    }
    return ret;
}
```

Q_{in}-7B

```
private String cleanWord(String word) {
    // Remove everything except alphabetic characters
    word = word.replaceAll("[^a-zA-Z']", "");

    // Convert to lowercase
    word = word.toLowerCase();

    // Remove leading/trailing whitespace
    word = word.trim();

    // If there's an apostrophe, keep only the part before it
    int index = word.indexOf('\'');
    if (index != -1) {
        word = word.substring(0, index);
    }

    return word;
}
```

Fig. 5 An example of disagreement among the evaluators on the ranking of the candidate improvements

Based on human evaluation, however, GPT-4.1 improves readability in ∼69% of the cases, and represents the best model according to all evaluators. Researchers should devise new approaches to automatically assess whether a candidate actually improves code readability.

To better understand the prompt impact on GPT-4.1, we performed additional experiments using two prompting-techniques. First, we employed an in-context learning (ICL) (Brown et al. 2020) prompt including a specialized system prompt, context, definition, and positive and negative implications of code readability, other than the actual instruction. We call this model G_{ICL}. Second, we extended the in-context learning prompt leveraging Few-shot learning (FS) (Ahmed and Devanbu 2022; Brown et al. 2020) using three examples (original code, improvement) extracted from the RQ_3 assessments. To select them, we first collected a preliminary pool of instances that all the evaluators assessed as improvement and, then, we selected the instances with the minimum, median, and maximum edit distances (Levenshtein 1966) with the aim of showing the model different magnitude of edits. We call this model G_{ICL+FS}. The prompts used are reported in the replication package (Anonymous 2025). We compared GPT-4.1 (G_{zs}) and G_{ICL} on the full test set, and G_{zs} and G_{ICL+FS} excluding the instances used in the shots for G_{ICL+FS}. The results show that G_{ICL} achieves an EM of 0.06%, halving G_{zs} EM (0.13%), while G_{ICL+FS} achieves an EM of 0.38 improving three times over G_{zs}. Since such metrics are not sufficient alone, E2 and E3 performed a manual analysis as described in RQ_3 on 50 random sampled instances from the test set comparing the three models candidates. Qualitatively speaking, the evaluators noted that the candidate improvements generated by the three models were actually very similar. Indeed, E2 reports 21/50 ties between G_{zs} and G_{ICL} and 22/50 ties between G_{zs} and G_{ICL+FS}, while E3 reports 21/50 ties between G_{zs} and G_{ICL} and 23/50 ties between G_{zs} and G_{ICL+FS}. Additionally, even G_{ICL} and G_{ICL+FS} obtained 24 and 25 ties from E2 and E3, respectively. On the other hand, E2 and E3 exhibited different preferences. E2 tended to prefer the G_{zs} candidates over G_{ICL} and G_{ICL+FS} (23/50 and 21/50, respectively); instead, E3 favored G_{ICL} and G_{ICL+FS} over G_{zs} (22/50 and 20/50 respectively). We believe that the similarity among the candidates increased evaluators reliance on subtle differences for the assessment, ultimately showing their subjective preference.

Based on the analysis we performed, we observe that prompt-engineering practices, with respect to the zero-shot prompting, could both reduce (G_{ICL}) and improve (G_{ICL+FS}) the ability to emulate developers changes, while the manual assessments indicates that such practices may not highly impact the generated candidates and may introduce subtle variations that are differently perceived by individual evaluators.

6 Threats to Validity

Threats to Construct Validity In this work we ran a differentiated replication of the study by Vitale et al. (2023). Like in the previous work, it is possible that the dataset contains low-quality instances. We applied some modifications to the original pipeline proposed in that work (*e.g.,* we introduced new terms to filter out unrelated commits). While our manual validation confirmed that our pipeline allows us to construct a less noisy dataset than the one from the previous work, we still acknowledge the presence of a non-negligible (docu-

mented) percentage of wrong training samples. An additional threat regards the size of the training set. In our study we collected ∼25k instances available for fine-tuning, a limited size with respect to other works (Huang et al. 2025; Improta et al. 2025). However, there is evidence that even limited-size training data allows to achieve good effectiveness (Crupi et al. 2026; Ye et al. 2025). Enlarging such size could provide different results.

Another threat regards data leakage (López et al. 2024; Vitale et al. 2025b). It is possible that some instances of our dataset were included in the training data of the models employed in this work, potentially leading to inflated performance. However, since both model families were trained on large-scale public corpora, any potential leakage would likely affect all evaluated models (both zero-shot and fine-tuned) in a comparable manner. In addition, we can not completely exclude the possibility that the models were exposed during pre-training to readability-improving commits or related patterns, potentially benefiting from such exposure during evaluation. At the same time, the very low exact-match rates observed in RQ_1 show that the models are not simply reproducing previously seen changes. Moreover, if the observed results were driven by leakage or memorization of developer-written commits, we would expect the generated changes to more closely resemble those commits. This does not appear to be the case, as human evaluators mostly preferred the zero-shot generated readability improvements over the original developer-written changes. Therefore, even if some degree of leakage occurred, we expect its practical impact on the validity of our conclusions to be negligible.

Threats to Internal Validity The presence of human errors in the extensive manual evaluation conducted in this work can be considered as a possible threat. 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. The manual analysis conducted in RQ_2 may present human errors. To further strengthen such analysis, we considered all the instances that were directly compilable without further context (*e.g.,* other classes from the same project). This resulted in 87 pairs. Then, we generated test cases with EvoSuite (Fraser and Arcuri 2011) starting from the original code version with one minute timeout (mean branch coverage ∼68%). Then, we ran the same test cases for both the candidate code to spot behavioral changes. Note, indeed, that EvoSuite automatically adds oracles that reflect the behavior of the code for which the test cases had been generated (*i.e.,* the original code, in our case). We found no cases in which we assessed that the candidate improvement was not changing the behavior of the original code while EvoSuite tests show the opposite. In addition, EvoSuite revealed four cases with behavioral differences, which were also identified by our manual analysis. On the other hand, we found eight cases we assessed as behavioral changing for which generated tests did not reveal differences. To confirm that our evaluation was correct, we tried to manually write tests that showed the behavioral changes. Out of the eight cases, five of them were correctly set as behavior changing, while the remaining three were false positives in our initial assessment. In general, the automated validation largely confirmed our manual findings, with only three false positives identified (∼3%) and zero false negatives.

We did not aim to reach consensus for RQ_3, in which we assumed differences were not due to possible errors by the evaluators but rather to different views on code readability,

which is subjective. Another possible threat regards how we split the instances to define training, validation, and test sets. Unlike Vitale et al., who randomly split their instances, we enforced repository-level splitting by assigning all instances from the same repository to a single split. We did this to limit the risk of data leakage (López et al. 2024).

Another threat concerns the metrics adopted to evaluate model performance in RQ_1. We relied on Exact Match (EM) and CrystalBLEU as quantitative indicators of model effectiveness. However, EM represents a value for model's capability in *emulating developer changes*. Indeed, as observed in RQ_3, a non-exact match may still constitute a valid readability improvement. For instance, if the developer change renames a variable from `a` to `userCount`, while the model generates `totalUsers`, the prediction would be considered incorrect under EM despite clearly improving readability. In a similar way, while CrystalBLEU indicates the similarity between the generated code and the developer change, it remains a text-based metric and may assign similar scores to edits that differ substantially in terms of code readability. Note, however, that the main conclusions of the paper are based on the manual evaluation performed in RQ_2 and, most importantly, RQ_3.

Threats to External Validity We only run our study on Java code. Using different programming languages might provide different results. It is also possible that the our results are strictly dependent on the specific models we studied (*i.e.,* Qwen2.5-Coder and GPT-4.1). To mitigate this threat, we employed four variants of the Qwen2.5-Coder model (*i.e.,* 0.5B, 1.5B, 3B, and 7B). Still, different models (*e.g.,* Deepseek-Coder Guo et al. 2024a; Zhu et al. 2024a or Claude 3.7) or larger LLMs (*e.g.,* 14B and 32B) could result in different findings.

7 Conclusion

We presented a differentiated replication of the work by Vitale et al. (2023) in the context of the modern LLM landscape. Our results generally confirms what the previous work suggested, *i.e.,* that LLMs can improve code readability. However, we provide evidence that fine-tuning on the set of developer readability changes we collected is not needed to improve readability. Our qualitative analysis clearly shows that LLMs in a zero-shot setting provide better recommendations than fine-tuned models. Even more interestingly and surprisingly, we observed that evaluators preferred the improvements suggested by LLMs in a zero-shot setting to the ones made by human developers. GPT-4.1, as a representative of closed-source zero-shot model, obtained the best results, followed by Qwen2.5-Coder 7B. Finally, we observed that no LLM is completely safe to use since they are all prone to change the behavior of the input code in at least one out of ten case. This suggests that practitioners should check the suggested improvements before integrating them.

We publicly release our dataset and all the scripts used to build it, together with the results of our manual investigations (Anonymous 2025).

Author Contributions Antonio Vitale: Developed the research idea, designed the study, conducted experiments, participated in the writing. Emanuela Guglielmi: Developed the research idea, designed the study, conducted experiments, participated in the writing. Valentina Piantadosi: Developed the research idea, designed the study, conducted experiments, participated in the writing. Antonio Mastropaolo: Developed the research idea, designed the study, conducted experiments, participated in the writing. Gabriele Bavota:

Developed the research idea, designed the study, participated in the writing. Rocco Oliveto: Developed the research idea, designed the study, participated in the writing. Simone Scalabrino: Developed the research idea, designed the study, participated in the writing.

Funding Open access funding provided by Politecnico di Torino within the CRUI-CARE Agreement. 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, "DevProDev: Profiling Software Developers for Developer-Centered Recommender Systems", grant n. 2022S49T4W, CUP: H53D23003610001, and "QualAI: Continuous Quality Improvement of AI-based Systems", grant n. 2022B3BP5S, CUP: H53D23003510006. Bavota acknowledges the financial support of the Swiss National Science Foundation for the PARSED project (SNF Project No. 219294).

Data Availability Statement We publicly release source code and data used in this work (Anonymous 2025).

Declarations

Ethical Approval Not applicable.

Informed Consent Not applicable.

Conflict of Interest The authors have no conflicts of interest to declare that are relevant to the content of this article.

Clinical trial number Not applicable.

References

Ahmed T, Devanbu P (2022) Few-shot training llms for project-specific code-summarization. In: Proceedings of the 37th IEEE/ACM International conference on automated software engineerin, pp 1–5

Allamanis M, Barr ET, Bird C, Sutton C (2014) Learning natural coding conventions. In: Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering, pp 281–293

AnonymousVitale A, Guglielmi E, Piantadosi V, Mastropaolo A, Bavota G, Oliveto R, Scalabrino S (2025) Replication package for "revisiting code readability improvements with llms: a replication and critical assessment of fine-tuned models". https://doi.org/10.5281/zenodo.16022058

Benjamini Y, Hochberg Y (1995) Controlling the false discovery rate: a practical and powerful approach to multiple testing. J Roy Stat Soc: Ser B (Methodol) 57(1):289–300

Brown T, Mann B, Ryder N, Subbiah M, Kaplan JD, Dhariwal P, Neelakantan A, Shyam P, Sastry G, Askell A et al (2020) Language models are few-shot learners. Adv Neural Inf Process Syst 33:1877–1901

Buse RP, Weimer WR (2008) A metric for software readability. In: Proceedings of the 2008 international symposium on Software testing and analysis, pp 121–130

Buse RP, Weimer WR (2009) Learning a metric for code readability. IEEE Trans Software Eng 36(4):546–558

Chen M, Tworek J, Jun H, Yuan Q, Pinto HPDO, Kaplan J, Edwards H, Burda Y, Joseph N, Brockman G (2021) Evaluating large language models trained on code. arXiv:2107.03374

Crupi G, Tufano R, Bavota G (2026) Improving code generation via small language model-as-a-judge. arXiv:2602.11911

Dorn J (2012) A general software readability model. MCS Thesis available from (http://www.cs.virginia.edu/weimer/students/dorn-mcs-paper.pdf) 5:11–14
Eghbali A, Pradel M (2022) Crystalbleu: precisely and efficiently measuring the similarity of code. In: Proceedings of the 37th IEEE/ACM international conference on automated software engineering, pp 1–12
Elkind E, Lang J, Saffidine A (2015) Condorcet winning sets. Soc Choice Welfare 44(3):493–517
Erlikh L (2000) Leveraging legacy system dollars for e-business. IT professional 2(3):17–23
Fisher RA et al (1956) Mathematics of a lady tasting tea. The world of mathematics 3(part 8):1514–1521
Fraser G, Arcuri A (2011) Evosuite: automatic test suite generation for object-oriented software. In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, pp 416–419
Grigorik I (2012) GitHub Archive. https://www.gharchive.org/
Guo D, Zhu Q, Yang D, Xie Z, Dong K, Zhang W, Chen G, Bi X, Wu Y, Li Y (2024a) Deepseek-coder: When the large language model meets programming-the rise of code intelligence. arXiv:2401.14196
Guo Q, Cao J, Xie X, Liu S, Li X, Chen B, Peng X (2024b) Exploring the potential of chatgpt in automated code refinement: An empirical study. In: Proceedings of the 46th IEEE/ACM international conference on software engineerin, pp 1–13
Huang K, Zhang J, Meng X, Liu Y (2025) Template-guided program repair in the era of large language models. In: ICSE, pp 1895–1907
Hui B, Yang J, Cui Z, Yang J, Liu D, Zhang L, Liu T, Zhang J, Yu B, Lu K et al (2024) Qwen2. 5-coder technical report. arXiv preprint arXiv:2409.12186
Improta C, Tufano R, Liguori P, Cotroneo D, Bavota G (2025) Quality in, quality out: Investigating training data's role in ai code generation. arXiv:2503.11402
Javaparser: Javaparser. https://javaparser.org/
Kavathekar I, Donakanti R, Kumaraguru P, Vaidhyanathan K (2025) Small models, big tasks: An exploratory empirical study on small language models for function calling. In: Proceedings of the 29th international conference on evaluation and assessment in software engineering, pp 1117–1126
Krishnamoorthy MS, Raghavachari M (2005) Condorcet winner probabilities-a statistical perspective
Levenshtein VI (1966) Binary codes capable of correcting deletions, insertions, and reversals. Soviet physics doklady, Soviet Union 10:707–710
Li Z, Lu S, Guo D, Duan N, Jannu S, Jenks G, Majumder D, Green J, Svyatkovskiy A, Fu S (2022) Automating code review activities by large-scale pre-training. In: Proceedings of the 30th ACM joint european software engineering conference and symposium on the foundations of software engineering, pp 1035–1047
Li J, Fang A, Smyrnis G, Ivgi M, Jordan M, Gadre SY, Bansal H, Guha E, Keh SS, Arora K et al (2024) Datacomp-lm: In search of the next generation of training sets for language models. Adv Neural Inf Process Syst 37:14200–14282
Lin B, Scalabrino S, Mocci A, Oliveto R, Bavota G, Lanza M (2017) Investigating the use of code analysis and nlp to promote a consistent usage of identifiers. In: IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM), pp 81–90. IEEE
Lin B, Wang S, Wen M, Chen L, Mao X (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, pp 1554–1566
López JAH, Chen B, Saad M, Sharma T, Varró D (2024) On inter-dataset code duplication and data leakage in large language models. IEEE Trans Software Eng
Loriot B, Madeiral F, Monperrus M (2019) Styler: Learning formatting conventions to repair checkstyle errors. arXiv:1904.01754
Loshchilov I, Hutter F (2017) Decoupled weight decay regularization. arXiv:1711.05101
Lozhkov A, Li R, Allal LB, Cassano F, Lamy-Poirier J, Tazi N, Tang A, Pykhtar D, Liu J, Wei Y et al (2024) Starcoder 2 and the stack v2: The next generation. arXiv:2402.19173
Mastropaolo A, Aghajani E, Pascarella L, Bavota G (2021) An empirical study on code comment completion. IN: 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp 159–170. IEEE
Mi Q, Keung J, Xiao Y, Mensah S, Gao Y (2018) Improving code readability classification using convolutional neural networks. Inf Softw Technol 104:60–71
Mi Q, Hao Y, Ou L, Ma W (2022) Towards using visual, semantic and structural features to improve code readability classification. J Syst Softw 193:111454
Mi Q, Zhan Y, Weng H, Bao Q, Cui L, Ma W (2023) A graph-based code representation method to improve code readability classification. Empir Softw Eng 28(4):87
Minelli R, Mocci A, Lanza M (2015) I know what you did last summer-an investigation of how developers spend their time. In: IEEE 23rd international conference on program comprehension, pp 25–35. IEEE

Oliveira D, Santos R, de Oliveira B, Monperrus M, Castor F, Madeiral F (2024) Understanding code understandability improvements in code reviews. IEEE Trans Softw Eng
OpenAI: GPT4.1. https://platform.openai.com/docs/models/gpt-4.1
Papineni K, Roukos S, Ward T, Zhu WJ (2002) Bleu: a method for automatic evaluation of machine translation. In: Proceedings of the 40th annual meeting of the Association for Computational Linguistics, pp 311–318
Penedo G, Kydlíček H, Lozhkov A, Mitchell M, Raffel CA, Von Werra L, Wolf T et al (2024) The fineweb datasets: Decanting the web for the finest text data at scale. Adv Neural Inf Process Syst 37:30811–30849
Piantadosi V, Fierro F, Scalabrino S, Serebrenik A, Oliveto R (2020) How does code readability change during software evolution? Empir Softw Eng 25(6):5374–5412
Posnett D, Hindle A, Devanbu P (2011) A simpler model of software readability. In: Proceedings of the 8th working conference on mining software repositories, pp 73–82
Prechelt L (2002) Early stopping-but when? Neural Networks: Tricks of the trade, pp 55–69. Springer
Prenner JA, Robbes R (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, pp 1–13
Raffel C, Shazeer N, Roberts A, Lee K, Narang S, Matena M, Zhou Y, Li W, Liu PJ (2019) Exploring the limits of transfer learning with a unified text-to-text transformer. arXiv:1910.10683
Ren H, Zhan M, Wu Z, Zhou A, Pan J, Li H (2025) Reflectioncoder: Learning from reflection sequence for enhanced one-off code generation. In: Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pp 9999–10020
Roziere B, Gehring J, Gloeckle F, Sootla S, Gat I, Tan XE, Adi Y, Liu J, Sauvestre R, Remez T (2023) Code llama: Open foundation models for code. arXiv:2308.12950
Saari DG, Merlin VR (1996) The copeland method: I.: Relationships and the dictionary. Econ Theor 8:51–76
Scalabrino S, Linares-Vasquez M, Poshyvanyk D, Oliveto R (2016) Improving code readability models with textual features. In: 2016 IEEE 24th International Conference on Program Comprehension (ICPC), pp 1–10. IEEE
Scalabrino S, Linares-Vásquez M, Oliveto R, Poshyvanyk D (2018) A comprehensive model for code readability. J Softw Evol Process 30(6):e1958
Sergeyuk A, Lvova O, Titov S, Serova A, Bagirov F, Bryksin T (2024a) Assessing consensus of developers' views on code readability. arXiv:2407.03790
Sergeyuk A, Lvova O, Titov S, Serova A, Bagirov F, Kirillova E, Bryksin T (2024b) Reassessing java code readability models with a human-centered approach. In: Proceedings of the 32nd IEEE/ACM international conference on program comprehension, pp 225–235
Shi E, Wang Y, Du L, Chen J, Han S, Zhang H, Zhang D, Sun H (2022a) On the evaluation of neural code summarization. In: Proceedings of the 44th international conference on software engineering, pp 1597–1608
Shi L, Mu F, Chen X, Wang S, Wang J, Yang Y, Li G, Xia X, Wang Q (2022b) 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, pp 107–119
Silva A, Fang S, Monperrus M (2025) Repairllama: Efficient representations and fine-tuned adapters for program repair. IEEE Trans Software Eng
Sun W, Miao Y, Li Y, Zhang H, Fang C, Liu Y, Deng G, Liu Y, Chen Z (2024) Source code summarization in the era of large language models. arXiv preprint arXiv:2407.07959
Thies A, Roth C (2010) Recommending rename refactorings. In: Proceedings of the 2nd international workshop on recommendation systems for software engineering, pp 1–5
Tufano R, Pascarella L, Bavota G (2023) Automating code-related tasks through transformers: The impact of pre-training. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), pp 2425–2437. IEEE
Vitale A, Piantadosi V, Scalabrino S, Oliveto R (2023) Using deep learning to automatically improve code readability. In: 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp 573–584. IEEE
Vitale A, Guglielmi E, Oliveto R, Scalabrino S (2025a) Personalized code readability assessment: Are we there yet?. arXiv:2503.07870
Vitale A, Oliveto R, Scalabrino S (2025b) A catalog of data smells for coding tasks. ACM Trans Softw Eng Methodol 34(4):1–32
Wang J, Huang Y, Chen C, Liu Z, Wang S, Wang Q (2024) Software testing with large language models: Survey, landscape, and vision. IEEE Trans Software Eng 50(4):911–936
WangJ, Xie X, Hu Q, Liu S, Yu J, Kong J, Li Y (2025a) Defects4c: Benchmarking large language model repair capability with c/c++ bugs. arXiv:2510.11059

Wang P, Zhang L, Liu F, Zhu Y, Xu W, Shi L, Lian X, Li M, Shen B, Fu A (2025b) Efficientedit: Accelerating code editing via edit-oriented speculative decoding. arXiv:2506.02780
Wang Q, Sun Z, Wang R, Huang T, Jin Z, Li G, Lyu C (2025c) Semguard: Real-time semantic evaluator for correcting llm-generated code. arXiv:2509.24507
Ye Y, Huang Z, Xiao Y, Chern E, Xia S, Liu P (2025) Limo: Less is more for reasoning. arXiv:2502.03387
Zhang J, Nie P, Li JJ, Gligoric M (2023) Multilingual code co-evolution using large language models. In: Proceedings of the 31st ACM joint european software engineering conference and symposium on the foundations of software engineering, pp 695–707
Zhu Q, Guo D, Shao Z, Yang D, Wang P, Xu R, Wu Y, Li Y, Gao H, Ma S (2024a) Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. arXiv:2406.11931
Zhu Q, Liang Q, Sun Z, Xiong Y, Zhang L, Cheng S (2024b) Grammart5: Grammar-integrated pretrained encoder-decoder neural model for code. In: Proceedings of the IEEE/ACM 46th international conference on software engineering, pp 1–13
Zhu T, Li Z, Pan M, Shi C, Zhang T, Pei Y, Li X (2024c) Deep is better? an empirical comparison of information retrieval and deep learning approaches to code summarization. ACM Trans Softw Eng Methodol 33(3):1–37

Authors and Affiliations

Antonio Vitale[1] · Emanuela Guglielmi[2] · Valentina Piantadosi[2] · Antonio Mastropaolo[3] · Gabriele Bavota[4] · Rocco Oliveto[2] · Simone Scalabrino[2]

✉ Antonio Vitale
antonio.vitale@polito.it; antonio.vitale@unimol.it

Emanuela Guglielmi
emanuela.guglielmi@unimol.it

Valentina Piantadosi
valentina.piantadosi@unimol.it

Antonio Mastropaolo
amastropaolo@wm.edu

Gabriele Bavota
gabriele.bavota@usi.ch

Rocco Oliveto
rocco.oliveto@unimol.it

Simone Scalabrino
simone.scalabrino@unimol.it

[1] Politecnico di Torino, Torino, Italy

[2] University of Molise, Termoli, Italy

[3] William & Mary, Williamsburg, VA, USA

[4] Università della Svizzera italiana, Lugano, Switzerland