LLaMaDoc: Intelligent Documentation Maintenance

A Smart Solution for Maintaining Accurate Code Documentation Using Language Models

Lara Kursawe and Christian Raue

Artificial Intelligence in Software Engineering
Hasso-Plattner-Institute

23.07.2024

Code updates often outpace corresponding documentation changes, resulting in outdated docstrings either due to time constraints or by mistake. Manually finding and updating the documentation can be a tedious and lengthy process. We present a VSCode extension that automatically identifies and updates outdated docstrings using a local large language model. We defined a metric to measure the quality of the out-of-date check and optimised the best parameters. Additionally, we fine-tuned the model using data scraped from GitHub repositories and evaluated the results by comparing the updated docstrings from the fine-tuned model with those from the pre-trained model, using GPT models to rate them. Our evaluation showed that the fine-tuned model did not produce better docstrings than the pre-trained model and that both models were not capable of reliably detecting and updating docstrings correctly. However, user testing showed very positive interactions with the extension. With more time and resources, the underlying model can be further updated and improved.

1 Introduction

When writing or developing code, it is common to adapt and update it regularly, while neglecting to update the respective code documentation. This might happen due to time constraints or simply because of an oversight. Normally, a developer would either have to review and update the entire code themselves at the end of a day, or manually instruct a language model to update it according to the new code. This is a tedious and time-consuming process.

Ideally, there would be a tool to automatically identify and update outdated documentation in a file. Such a tool would not only save developers significant time but also highlight any documentation that they might otherwise have overlooked.

Recent advances in large language models (LLMs) have made them a promising candidate for the development of such a tool. AI has the ability to automate cognitive tasks and can therefore reliably detect and update code snippets. While

traditional methods might be able to track past code changes, they lack the flexibility and adaptability of LLMs in such an open context.

Our goal is to build this tool and to provide an easy-to-use, minimalistic interface for users, delivered via a VSCode extension. The extension should detect all outdated docstrings in a Python file and allow the user to update them individually.

To achieve this, we plan to fine-tune an existing model specifically for Python code and outdated code documentation. While providing support for additional programming languages would be nice, it is out of scope for this project. The model we use should be reliable and accurately identify out-of-date documentation, with neither false positives (indicating a docstring is out-of-date when it is not) nor false negatives (failing to detect an out-of-date docstring), but still have an acceptable runtime on the provided hardware. So we must find a balance between an acceptable latency and accuracy for the detection and updates of the docstrings.

We therefore present *LLaMaDoc*[1], a VSCode extension designed to streamline and automate the process of updating outdated code documentation. Our extension utilises JavaScript for the VSCode plugin and Python for detecting and updating of out-of-date docstrings using the `google/codegemma-2b` model, as well as for the interactions between the model and the extension.

To obtain training data for fine-tuning the model, we scraped GitHub for repositories and filtered them based on various metrics.

While the plugin offers a user-friendly interface and functions well, the underlying model does not meet our expectations in terms of performance. It can accurately detect and update docstrings for simple functions, but it struggles with inconsistencies in naming and more complex algorithms. This performance could have been improved with better hardware and more time for fine-tuning. If resources become available, upgrading to a more powerful model and re-running the fine-tuning process would be fairly simple.

2 Related Work

The problem we addressed has been explored by several researchers. Here we give a brief overview of some of the approaches that have previously been studied.

Sadu [3] used heuristics and a classical algorithmic approach, relying on information from the code's syntax tree and semantic matching to link code changes with necessary updates to docstrings. This method showed promising results, though it was evaluated on only 30 examples and is language-dependent, working solely with Java code.

Another classical algorithm-based method was proposed by Huang et al. [2], which extracts features from code and docstrings using the abstract syntax tree and natural language processing. A random forest classifier then determines whether a docstring is outdated. This approach also demonstrated good results, and it is

[1] `https://github.com/CR1337/LLaMaDoc`

expected that more complex models, like transformer-based ones, could offer even further improvements.

Liu et al. [1] used a transformer-based model, similar to our approach, to generate updated docstrings. Their method aligns closely with our own.

Additionally, a consumer product *Docify AI*[2] in the form of a VSCode Extension exists, which generates docstrings and other documentation for code. This indicates that our choice to develop a VSCode extension is sound, as such extensions appear to be popular and in demand.

3 Approach

Our system mainly consists of two components, as shown in Figure 4. The first component is the VSCode extension itself, which the user interacts with. The second component, where the "heavy lifting" occurs, is the backend running on a server equipped with the necessary hardware needed to efficiently handle a large language model.

3.1 Extension Development

Initially, a user can scan the document in the active editor for outdated docstrings. Once the scan is complete, outdated docstrings are tagged as "Outdated Docstring", and a lightbulb icon appears in the same line when the cursor is placed on the function definition. This lightbulb provides various VSCode actions if possible, along with an option from our extension to either update the docstring for the corresponding function or dismiss the outdated tag. Users can also choose to dismiss all outdated docstrings in the current file.

3.1.1 Finding Outdated Docstrings

For the detection of outdated docstrings, we used two different transformer-based language models. `google/codegemma-2b`[3] is a large language model designed for code generation and completion. Since its training data included docstrings, we utilised this model to generate docstrings that match the given code.

Initially, we attempted a generation-based approach, by calculating the geometric mean of all docstring token probabilities computed by the LLM for both the user's docstring and the generated docstring. Unfortunately, this proved to be too slow in practise and we did not pursue this method further. Instead, we decided to use the `microsoft/codebert-base`[4] model for an embedding-based approach. The model embeds both code and docstrings, allowing us to compare them within the embedding space.

When initiating a scan, the extension first clears any previous actions and decorations from the editor. The source code and docstrings for each function are collected and sent to the backend. The `google/codegemma-2b` model generates

[2] `https://www.ai4code.io`, last accessed: 21.07.24

[3] `https://huggingface.co/google/codegemma-2b`, last accessed: 21.07.24

[4] `https://huggingface.co/microsoft/codebert-base`, last accessed: 21.07.24

a new docstring for each function without knowledge of the user's docstring, while the `microsoft/codebert-base` model embeds the source code, the user's docstring, and the generated docstring into a 767-dimensional vector space. The vectors are normalised, and the Euclidean distance between each docstring and its corresponding source code is calculated. The ratio of the distance between the code and the user's docstring to the distance between the code and the generated docstring is then computed. If the ratio exceeds the empirically determined threshold of 1.1, the user's docstring is considered out-of-date. This procedure is performed for each docstring sent by the extension, and the results are returned to be displayed to the user, as seen in Figure 6. The result of a scan is parsed as JSON and includes information such as the line numbers for the beginning and end of each function, their corresponding docstrings, and their outdated-status. If a docstring is outdated, the extension tags the function and generates a lightbulb using the VSCode Code Action Provider, as shown in Figure 1.

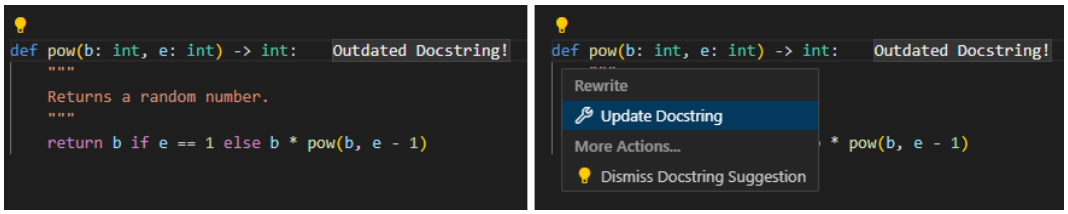

Figure 1: The user interface to update a docstring.

3.1.2 Updating Outdated Docstrings

When the user wants to update a specific docstring, they can instruct the extension to send the corresponding source code to the backend to request an updated version of the docstring. The backend creates a prompt for the LLM to generate a docstring for the specified function, as seen in Figure 5. For example, the following `pow` function would be turned into a prompt like this:

```
def pow(b: int, e: int) -> int:
    return b if e == 1 else b * pow(b, e - 1)
```

```
<|fim_prefix|>def pow(b: int, e: int) -> int:
    """<|fim_suffix|>"""
    return b if e == 1 else b * pow(b, e - 1)<|fim_middle|>
```

This prompt utilises the model's capability for fill-in-the-middle (FIM) semantics. The model generates tokens it expects to replace the `<|fim_suffix|>` token after the `<|fim_middle|>` token, allowing it to fill in a docstring that matches the surrounding code. The generated docstring is then returned to the extension and the old docstring in the document is replaced with the new one. If the new docstring has a different number of lines compared to the old one, the line numbers of subsequent docstrings and functions are adjusted accordingly, along with any decorations and

lightbulbs for functions below the current one to reflect the new line positions. The outdated docstring tag and lightbulb for the updated line are cleared and the user is notified of the successful update.

3.2 Large Language Model Development

To improve the model's performance for our specific use case, specifically, distinguishing between correct and outdated docstrings and updating them accordingly, we decided to fine-tune it. This process required us to first collect relevant and representative data.

3.2.1 Data Collection

We wanted to obtain real-world data, so we scraped code from GitHub using the GitHub Repository Search. Since we only wanted to handle Python code, we first filtered for repositories containing content written in Python. Next, we needed to ensure the quality of the code and the corresponding docstrings was sufficient. Due to budget constraints, we could not use an automated approach with one of OpenAI's GPT models to fulfil this requirement. Instead, we decided to restrict our search to repositories with more than 1.000 stars, assuming that more popular repositories would generally have better quality code and docstrings. This process resulted in a total of 895 scraped repositories, whose properties are shown in Figure 7.

All properties, except for the number of contributors, follow a power law distribution. The number of contributors appears to follow a normal distribution centered around a mean of 30, but with a very flat and heavy tail to the left. We considered removing repositories with a low number of contributors, operating under the assumption that high-quality docstrings become more important as more people work on a project. Ultimately, we decided against this for two reasons. First, there were only a few repositories with a low number of contributors, so removing them would not significantly reduce the overall data. Second, we wanted to ensure our data represented repositories with both many and few contributors. For all other properties, we did not see a need to further restrict the data.

Next, we extracted all Python functions with their respective docstrings across all versions, treating each change to a function in the entire commit history as a separate data point. Manual inspection of the data revealed that the initial version of a function often contained a docstring but no code. Including these in the training data would train the model to hallucinate docstrings for empty functions, so we removed the first version of any function. After this filtering step, we had approximately 5.2 million data points remaining.

To create training and test datasets, we randomly sampled from all data points, resulting in a training dataset of $40,000$ samples and a test dataset of $9,266$ samples. As the data only contained samples where the docstring fits the code, we modified the test data by randomly reassigning the docstrings to other data points, creating $9,266$ samples with non-matching docstrings. Lastly, we combined both sets to form a final test dataset of $18,532$ samples.

3.2.2 Fine-Tuning

We used these datasets to fine-tune the `google/codegemma-2b` model. Our fine-tuning method closely followed the example presented in class[5]. After tokenisation, we split our training data into blocks of 512 tokens. To avoid unwanted results from splitting a code example across two blocks, we ensured that each example fit entirely within a block. If an example didn't fit, we padded the block with `[EOS]` tokens and placed the example in the next block. Fine-tuning was conducted using Low-Rank Adaptation (LoRA) with a rank of 8. We used a batch size of 4 and ran the process for 10 epochs, using a learning rate of 0.0003. A larger batch size was not feasible due to the limitations of our local hardware. Due to the project's time constraints, we could only run the fine-tuning once and did not have the opportunity to adjust and optimise the parameters further.

4 Evaluation & Discussion

We state our research questions as follows:

- "Is it possible to automatically detect and update outdated docstrings for Python functions using a local[6] large language model?"
- "Does fine-tuning the model improve the quality of the results?"

To address these questions, we conducted several evaluations.

4.1 Studies

We carried out three studies to evaluate our results. First, a quantitative study assessed the performance of our out-of-date docstring test. Second, a quali-quantitative study using `GPT-3.5` and `GPT-4o` evaluated how effectively our system generates updated docstrings. Lastly, a small qualitative user study provided insights into the way users interact with our extension.

4.1.1 Evaluation of the Out-of-Date Test

Before evaluating our out-of-date test procedure, we first had to fine-tune several parameters to their optimal values. Specifically, these parameters included the **threshold** that separates distance ratios from indicating a docstring is out-of-date or not, the **model** to use, the **distance metric** to apply, the choice of whether to **normalise** the distance vectors, and the decision to take **multiple samples** for the generated docstring.

To achieve this, we explored all possible combinations of the categorical parameters. For the threshold values, we tested 200 equally spaced points between 0 and 3. We used a randomly sampled subset of 512 datapoints from the test data because of time constraints.

[5] Hasso-Plattner-Institute, AI in Software Engineering, 14.05.2024.

[6] In this context, *local* indicates that we are hosting the LLM on our own server, without relying on third-party services.

Since our procedure functions as a binary classifier, we initially used the F1-Score to measure performance. However, we found that the F1-Score is not ideal for optimisation tasks like this one, as it does not account for true negatives (TN) and thus only maximises true positives (TP). An appropriate alternative is the P4-Score:

$$P_4 = \frac{4 \cdot TP \cdot TN}{4 \cdot TP \cdot TN + (TP + TN) \cdot (FP + FN)} \tag{1}$$

FP and FN meaning false positives and false negatives respectively. The results are shown in Figure 8, the maximum values for all curves, along with their true and false positives and negatives, are detailed in Table 1. They suggest that the optimal combination of parameters is using multisampling with the fine-tuned model, the Euclidean distance metric, and a threshold of 1.39. This configuration yields a P4-Score of 0.53, a precision of 0.56 and a recall of 0.69, which can be interpreted as a moderately good performance.

Since the optimised parameters depend on the data used for tuning, we conducted an additional evaluation with independent data. We took another random sample of the same size from the test data for this purpose. This evaluation resulted in a P4-Score of 0.34 with 212 true positives, 39 true negatives, 220 false positives and 41 false negatives. This corresponds to a precision of 0.49 and a recall of 0.84. Although these results are not ideal, they are preferable to having a lower recall value in practise. From a user perspective, the high number of false positives means that up-to-date docstrings may be incorrectly flagged as outdated, prompting users to generate new ones. According to the high recall value, our model captures most of the positive examples and does not make many false negative errors. This is preferable to missing out on detecting actual outdated docstrings, as it ensures that potential issues are still addressed.

4.1.2 Evaluation of Docstring Updates

To evaluate the updated docstrings produced by our models, particularly the fine-tuned model, we used AI as a judge. We randomly sampled 100 code examples with outdated docstrings from our dataset and generated updated docstrings using both models. We then asked `GPT-3.5` and `GPT-4o` to rate, which docstring is the most appropriate for each code example.

To yield better results, we instructed GPT to first explain its reasoning. It was then asked to decide whether the initial outdated docstring (*"outdated"*, which should never be chosen, as all examples had outdated docstrings), the updated docstring from the pre-trained model or the updated docstring from the fine-tuned model was the best fit for the code. If neither docstring was suitable, GPT was to indicate that no docstring was a good match (*"none fits"*).

The results shown in Figure 2 indicate that the fine-tuned model does not achieve the best performance. `GPT-3.5` rated the fine-tuned docstrings as most fitting in 21 cases, while `GPT-4o` did so in 19 out of 100 cases.

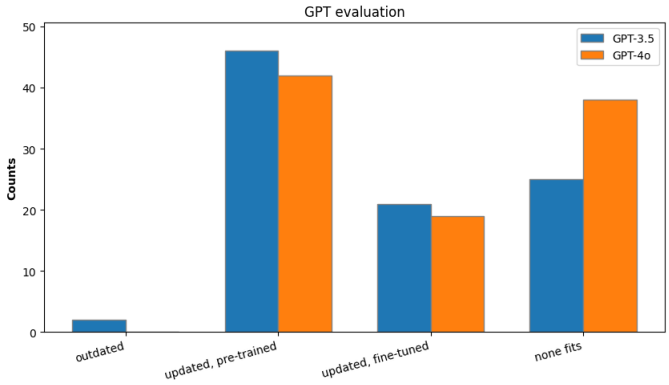

Figure 2: Evaluation of best fitting docstring updates using AI as a judge.

4.1.3 User Study

In our qualitative user study, we aimed to assess how users interact with the extension and whether it improves their everyday workflow. We introduced the extension to six participants, all of whom were either software developers or computer science students. After a brief overview of the plugin, we allowed them to explore it independently using a Python file with seven code examples. This file included a mix of outdated and correct docstrings for them to evaluate.

The feedback we received was mostly positive and included many suggestions for further developing and improving the extension. Some participants are not regular users of VSCode and mainly use JetBrains IDEs for software Development, so they were unfamiliar with the Quick Actions offered by the lightbulb. They suggested using coloured squiggly lines to highlight outdated docstrings instead. In contrast, those who frequently use VSCode immediately understood how to interact with it. Once introduced to the lightbulb feature however, all users found the interaction with the user interface straightforward and easy to use.

Most participants suggested providing a preview of the updated docstring or offering users a choice between two to three different options. They also recommended highlighting which parts of the docstring are incorrect or outdated and allowing users to give instructions to the model on how to update the docstring. While all of these are valuable suggestions, they were not feasible to implement within the project's time constraints. However, they are certainly worth exploring for future development of the plugin.

4.2 Results

We carried out three studies to evaluate different aspects of our project. After optimising the parameters for the out-of-date test, the best P4-Score was achieved using the fine-tuned model. However, when evaluating the updated docstrings, the fine-tuned model performed worse than the pre-trained model according to GPT-evaluations.

Upon manual inspection using the extension, it became evident that neither model performed as well as we had hoped. Unless very simple and clear examples

were used, the models struggled to differentiate between outdated and correct docstrings. The fine-tuned model often produced unhelpful or incorrect documentation, as seen in Figure 3.

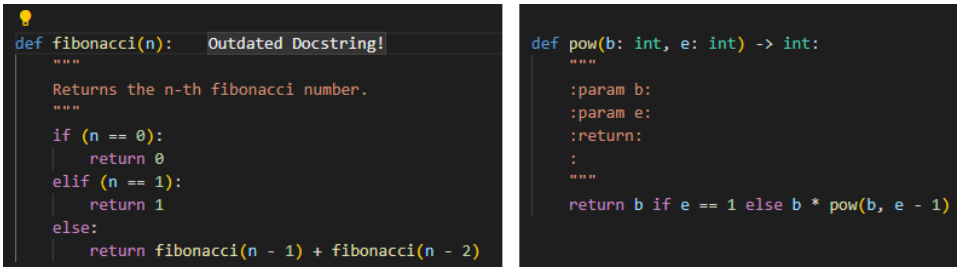

Figure 3: A wrongly tagged docstring by the pre-trained model and a nonsensical docstring update suggestion by the fine-tuned model.

Therefore, while we can answer our first research question — whether it is possible to automatically detect and update outdated docstrings for Python functions using a local large language model — with "yes," we must note that this applies primarily to very simple examples.

We must however answer our second research question — asking whether fine-tuning the model improves the quality of the results — with "no". In our case, fine-tuning resulted in worse quality for updated docstrings, which was confirmed by us and AI as a judge. Although this may be specific to our fine-tuning method and data, and not indicative of the general case.

4.3 Discussion

Overall, the underlying model we used for our plugin did not meet our expectations, as the generated docstrings were not helpful and the detection of outdated docstrings was inconsistent.

Fine-tuning the model did not improve the results; in fact, it made the suggested updates worse compared to the pre-trained model. This may be due to ineffective training, possibly caused by uncleaned data from the GitHub repositories or suboptimal fine-tuning parameters. We collected our own data because we wanted to incorporate mining GitHub repositories into our project. Although we aimed to select the best code examples possible, it was not feasible to manually check and clean the entire dataset. For that reason, we decided to switch back to the original pre-trained model in our extension, which has an optimal threshold of 1.1, as seen in Table 1. Another approach would be to use an existing dataset, such as `python_train`[7] from Kaggle. While further fine-tuning with better data could potentially improve performance, it may also be necessary to use a more powerful model, such as `google/codegemma-7b`. Alternatively, we could opt to not run the model locally and instead use the OpenAI API to access a much more sophisticated model, such as `GPT-4o`.

[7] `https://www.kaggle.com/datasets/elindbergs/python-train`, last accessed: 21.07.24

Nevertheless, our plugin received very good user feedback and serves as solid foundation for further development. When testing some examples with `GPT-4o`, it easily detected and updated all outdated docstrings without generating any false positives, as shown in Figure 9 and Figure 10. If more resources and time become available, we could simply swap out the model for a more powerful one and run the fine-tuning again.

4.3.1 Limitations

Since we aimed to cover a wide range of tasks during this project — such as configuring and fine-tuning an LLM ourselves, obtaining training data by scraping GitHub repositories, developing a functional VSCode extension, and evaluating our results — we were not able to dedicate as much time to each task as might be necessary to achieve optimal results. Our training data, in particular, could have benefited from more manual inspection to ensure quality, and the fine-tuning process would have been improved by multiple runs and optimised training parameters.

However, we were limited not only by time but also by resources. The provided GPU with 32GB memory was shared with another project group and often unavailable for hours at a time due to insufficient storage. When generating updated docstrings to evaluate the fine-tuned model, we were unable to generate more than 25 updates at a time without the server crashing.

These constraints limited our effectiveness, but we are still pleased with the results we achieved given the circumstances.

4.3.2 Future Work

By exchanging the underlying model used by our extension, it would immediately be a powerful tool in a software development workflow. Additional features such as discussed in Section 4.1.3 could further improve the user interface and turn *LLaMaDoc* into a valuable tool for VSCode users.

Expanding the application to support additional programming languages or adding functionality for Python comments within the code would also increase the applicability and utility of our extension further.

5 Conclusion

We developed and evaluated a VSCode extension *LLaMaDoc*[8], which automatically detects and updates Python code documentation using a local large language model. In order to improve performance, we fine-tuned the model using real-world data collected from GitHub. Although neither the pre-trained, nor the fine-tuned model met our performance expectations, we received positive feedback from users testing the extension. With additional time and resources, we could employ a more powerful model and turn *LLaMaDoc* into a valuable asset for Python developers.

[8] `https://github.com/CR1337/LLaMaDoc`

References

[1] S. L. et al. "TBCUP: A Transformer-based Code Comments Updating Approach". In: *IEEE* 47 (2023). https://ieeexplore.ieee.org/document/10196927, pages 892–897.

[2] Y. H. et al. "Are your comments outdated? Towards automatically detecting code-comment consistency". In: *arxiv* (Mar. 2024). https://arxiv.org/abs/2403.00251.

[3] A. Sadu. "Automatic detection of outdated comments in open source Java projects". https://oa.upm.es/56377/. Thesis. Universidad Politécnica de Madrid, 2019.

6 Appendix

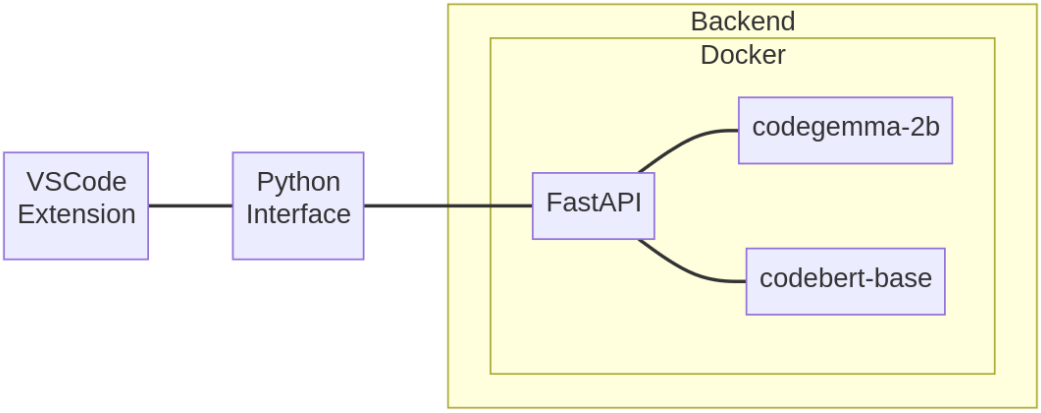

Figure 4: The coarse architecture of the system. The VSCode Extension communicates with a FastAPI backend utilizing the models codegemma-2b and codebert-base.

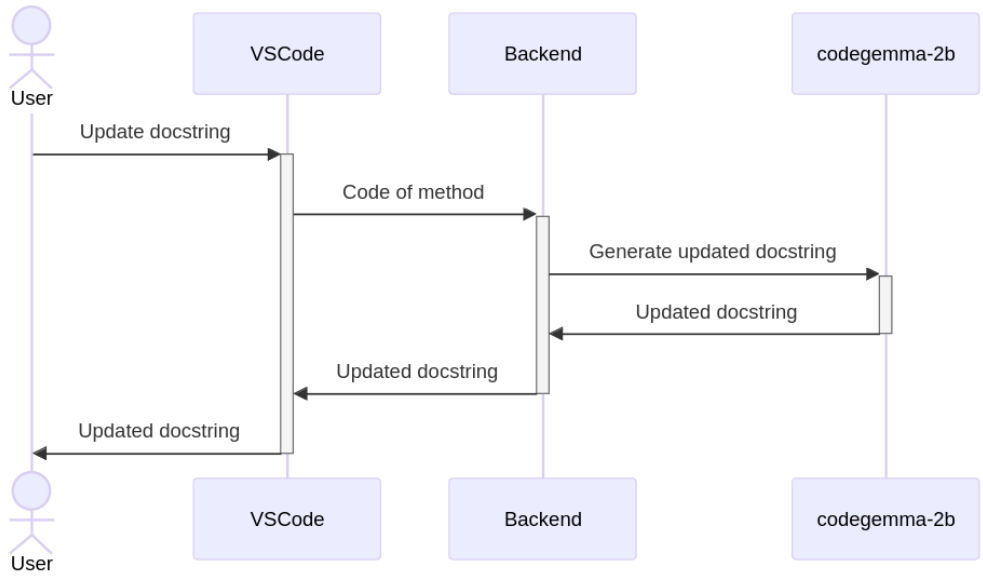

Figure 5: The interaction between a user and the system in order to update a single docstring.

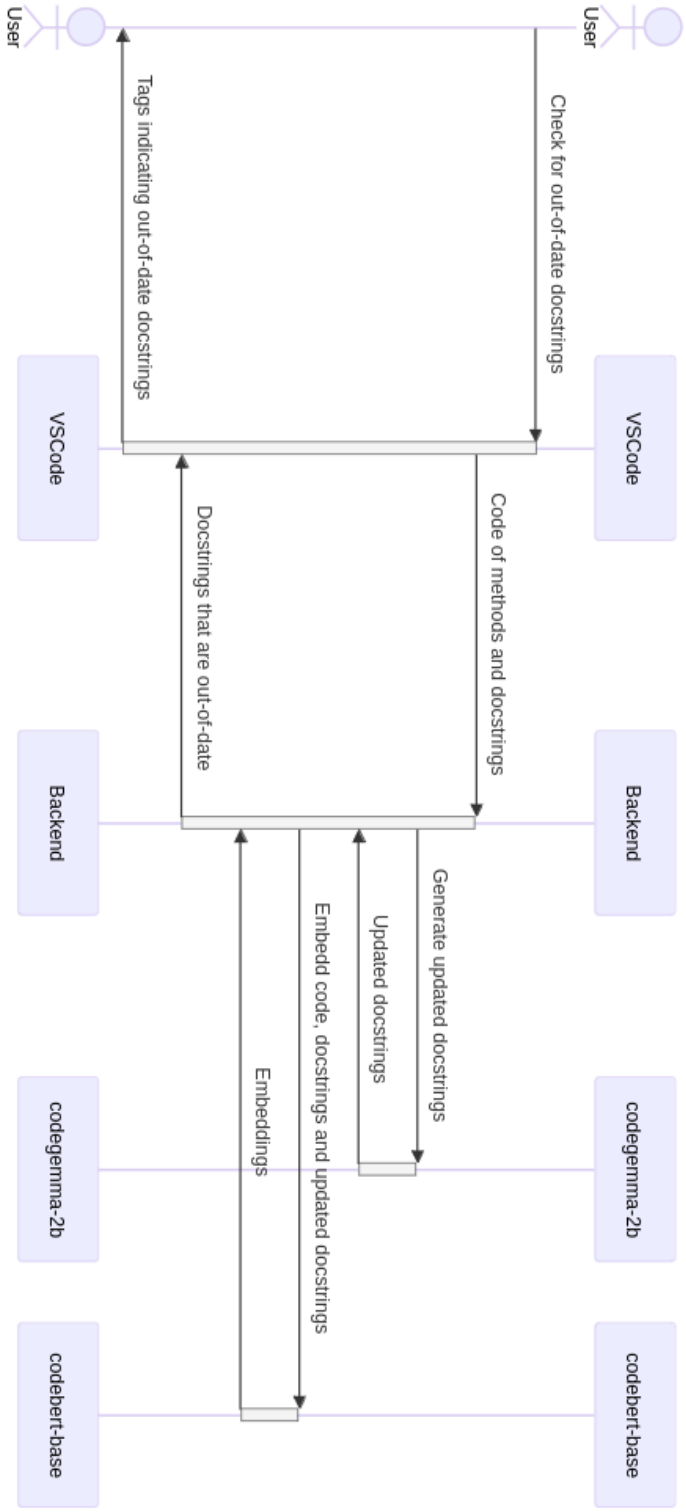

Figure 6: The interaction between a user and the system in order to check for out-of-date docstrings.

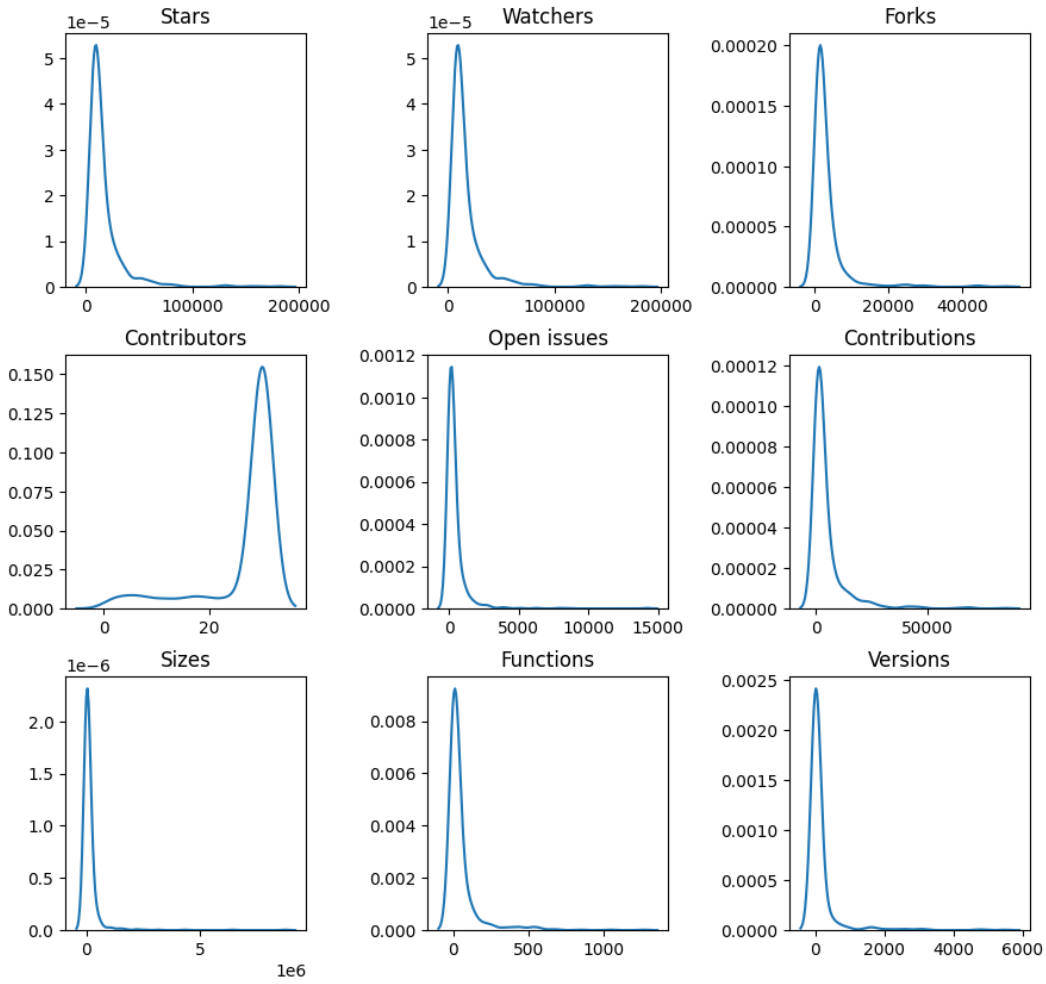

Figure 7: Distributions of Stars, Watchers, Forks, Contributors, Open Issues, Contributions, Size, Number of Functions, and Number of different Versions of Functions across all collected GitHub repositories.

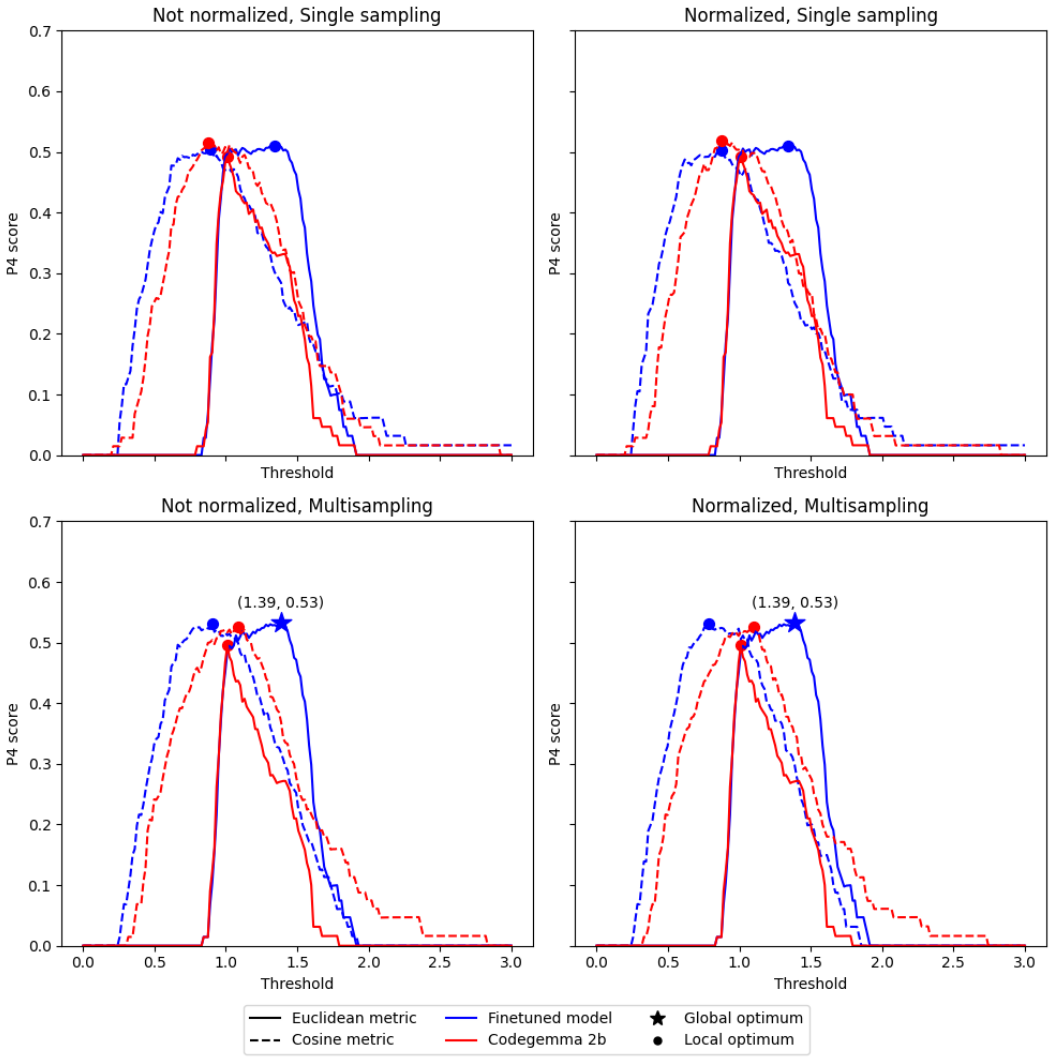

Figure 8: P4-Scores for various thresholds, comparing the ratios of distances between code and user docstrings to distance between code and generated docstrings. Scores were computed for all model combinations (pre-trained and fine-tuned).

#	Finetuned	Metric	normalized	Multisampling	Threshold	P_4	TP	TN	FP	FN
1	Yes	cosine	No	No	1.34	0.51	186	91	152	83
2	Yes	cosine	No	Yes	1.39	0.53	185	100	143	84
3	Yes	cosine	Yes	No	1.34	0.51	186	91	152	83
4	Yes	cosine	Yes	Yes	1.39	0.53	185	100	143	84
5	Yes	euclidean	No	No	0.89	0.50	139	120	123	130
6	Yes	euclidean	No	Yes	0.90	0.53	131	141	102	138
7	Yes	euclidean	Yes	No	0.87	0.50	139	119	124	130
8	Yes	euclidean	Yes	Yes	0.78	0.53	115	160	83	154
9	No	cosine	No	No	1.01	0.49	130	122	121	139
10	No	cosine	No	Yes	1.01	0.50	119	135	108	150
11	No	cosine	Yes	No	1.01	0.49	130	122	121	139
12	No	cosine	Yes	Yes	1.01	0.50	119	135	108	150
13	No	euclidean	No	No	0.87	0.51	114	152	91	155
14	No	euclidean	No	Yes	1.09	0.53	156	116	127	113
15	No	euclidean	Yes	No	0.87	0.52	115	153	90	154
16	No	euclidean	Yes	Yes	1.10	0.53	169	107	136	100

Table 1: Results of parameter tuning for the out-of-date test.

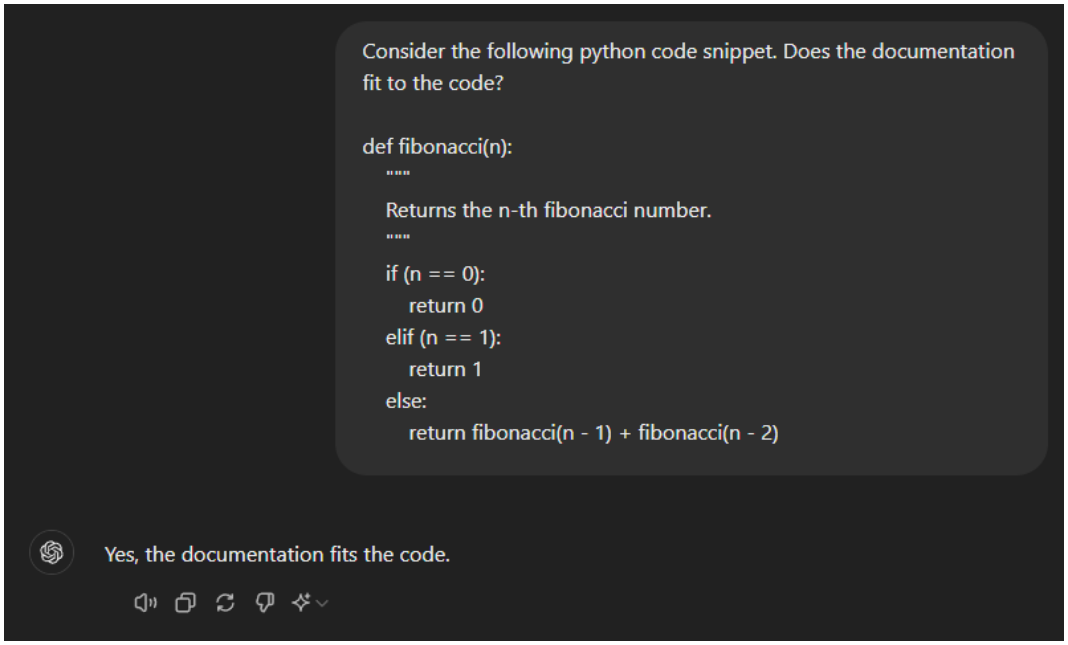

Figure 9: A docstring detection by ChatGPT.

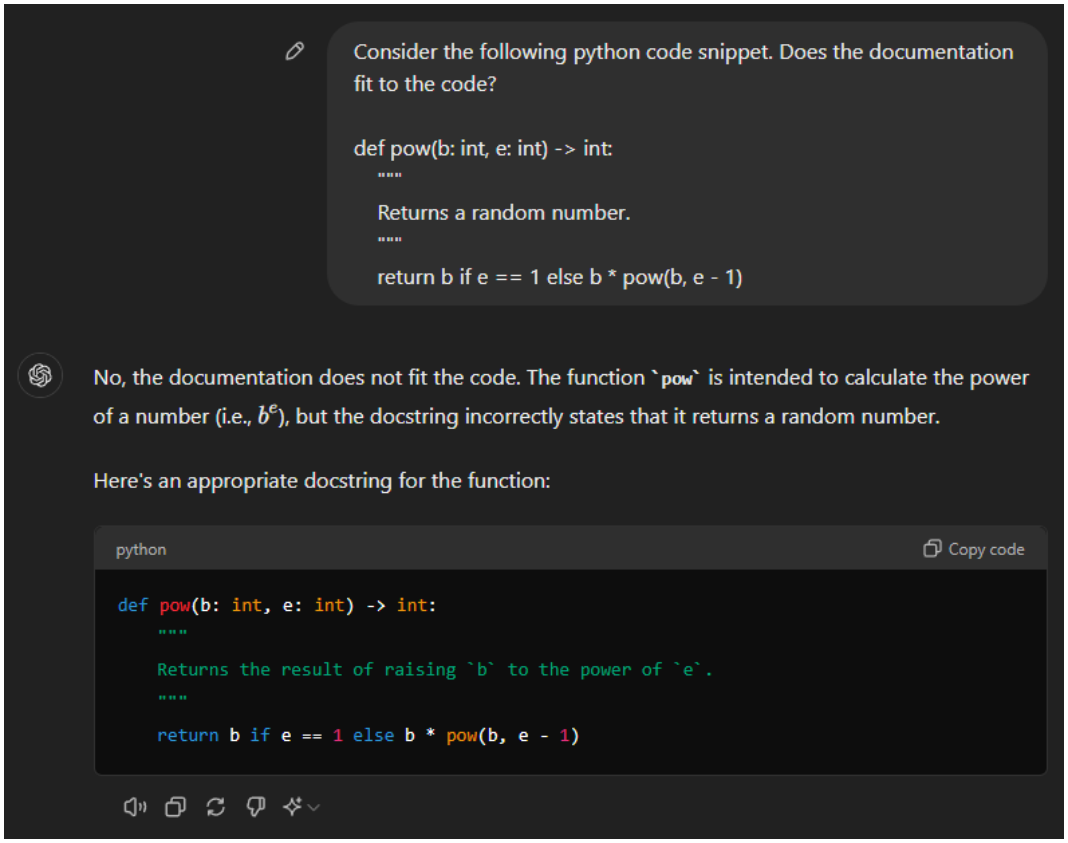

Figure 10: A docstring update by ChatGPT.