StRuCom: A Novel Dataset of Structured Code Comments in Russian

Maria Dziuba[1,2], Valentin Malykh[1,2,3],
[1]MTS AI, [2]ITMO University, [3]IITU University,
dziuba.maria@niuitmo.ru
valentin.malykh@phystech.edu

Abstract

Structured code comments in *docstring* format are essential for code comprehension and maintenance, but existing machine learning models for their generation perform poorly for Russian compared to English. To bridge this gap, we present StRuCom — the first large-scale dataset (153K examples) specifically designed for Russian code documentation. Unlike machine-translated English datasets that distort terminology (e.g., technical loanwords vs. literal translations) and docstring structures, StRuCom combines human-written comments from Russian GitHub repositories with synthetically generated ones, ensuring compliance with Python, Java, JavaScript, C#, and Go standards through automated validation.

1 Introduction

The automated generation of structured code comments in *docstring* format, including detailed descriptions of functionality, parameters, return values, exceptions, and usage examples, greatly improves codebase maintenance. Structured code comments provide developers with quick and easy access to the required information, and can also be used to automatically generate project documentation, for instance, in HTML format. However, modern language models, such as Qwen2.5-Coder (Hui et al., 2024) and DeepSeek-Coder (Guo et al., 2024), primarily focus on English-language data and therefore perform poorly for Russian-language comment, neglecting the needs of Russian-speaking developers. These developers, working on localized projects, who often encounter linguistic barriers, which can lead to code misunderstanding and a waste of time. In view of this, there is a strong need for a specialized model for this task, which requires curated training data.

Unfortunately, existing datasets (English-centric CodeSearchNet (Husain et al., 2019) or multilingual MCoNaLa (Wang et al., 2023b)) mostly focus on code summarization and retrieval tasks, not on function-level documentation generation. The datasets that contain both simple comments and docstrings in English (for example, the Vault (Nguyen et al., 2023)), firstly, require a tool for structure-based filtration to check comments for existence of detailed functionality descriptions, covering all function parameters, exceptions and its return value. Secondly, machine translation of English comments cannot be straightforwardly used, as it introduces distortions (Wang et al., 2023b) and disrupts *docstring* structure.

In this work, we present StRuCom, the first specialized dataset for generating structured Russian-language code comments. To create it, we developed a tool for filtering and validating comment structures, supporting five popular documentation styles: Python - GoogleDoc[1], JavaScript - JSDoc[2], Java - JavaDoc[3], C# - XML[4], and Go - GoDoc[5]. The dataset combines real-world comments from Russian repositories with synthetically generated examples. Using this data, we finetuned the Qwen2.5-Coder model family (0.5B, 1.5B, 3B, and 7B parameters), demonstrating statistically significant improvements in generation quality via `chrF++` (Popović, 2017) and `BERTScore` (Zhang et al.) metrics compared to baseline versions.

Our contributions: **Filtering tool for structured comments.** We developed an automated tool to validate comment structures across five documentation standards (Python, Java, Go, C#, JavaScript). **Dataset.** We compiled a dataset of 153K Russian-language code-comment pairs,

[1]https://google.github.io/styleguide/pyguide.html
[2]https://jsdoc.app
[3]https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html
[4]https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/xmldoc/recommended-tags
[5]https://tip.golang.org/doc/comment

combining real-world examples from GitHub repositories with synthetically generated annotations for five programming languages.

2 Related Work

The existing datasets for code-to-text tasks are mainly focused on English-language content. **The Stack** (Kocetkov et al., 2022) combines multilingual code from 658 programming languages (67 TB in version 2.x), collected from a variety of sources: Software Heritage Archive, GitHub Issues, Stack Overflow, etc. Despite its scale, the set is not adapted for supervised fine-tuning (SFT) tasks and requires significant preprocessing. **The Vault** (Nguyen et al., 2023), derived from The Stack v1, includes 43 million English-language code-text pairs from 10 programming languages. The data was obtained by extracting docstrings and inline comments using the *Code-Text* parser [6]. However, structured comments (with parameters and usage examples) remain rare, which is partly explained by the predominance of short functions in the source data. **CodeSearchNet** (Husain et al., 2019), part of the CodeXGLUE benchmark (Lu et al., 2021), contains 1 million English-language code-text pairs for 6 languages. The set is focused on code search: text descriptions are limited to the first paragraphs of the documentation, which simplifies comparison, but excludes complex descriptions. **MCoNaLa** (Wang et al., 2023b) offers limited multilingual support: 345 Russian, 341 Spanish, and 210 Japanese intent-snippet pairs for Python. The focus on narrow "how-to" scenarios and a small size limit the applicability of this dataset for structured documentation tasks.

3 StRuCom Dataset

Collection Process. To construct our dataset, we crawled all existing Russian-language repositories on GitHub for the selected programming languages (Python, Java, JavaScript (JS), C#, and Go). Since the GitHub API does not provide a direct query to identify the natural language used by repository authors, we developed a novel approach to address this limitation. Our program retrieved repositories with Russian-language descriptions and permissive licenses (allowing commercial use or lacking licensing restrictions). The crawled repositories contained comments written in various languages. For details on comment extraction see Appendix A.

Filtration Process. At the initial stage of filtering, all comments were standardized to follow a uniform style based on the conventions established for each programming language: Python - GoogleDoc, JavaScript - JSDoc, Java - JavaDoc, C# - XML, and Go - GoDoc. Examples of these standardized formats can be seen on Fig. 1. To further divide comments into types by structure, we suggest the following terminology: *A structured comment* is a comment that can be parsed by the `docstring_parser` library[7] and contains either parameter lists, return value descriptions, or exception descriptions. *A complete comment* is a structured comment that provides a comprehensive description of all its component parts, including types (if needed). *An incomplete comment* is a structured comment that lacks a description of any of its component parts, which is why it cannot be called complete. *Unstructured comments* are those that do not correspond to a specific format used in a given programming language. For more information about filtration by structure see Appendix D. Only structured and complete comments were included in the final version of the dataset.

Enhancement with LLM. Based on the statistics on the structuredness of the collected data from GitHub, many code comments are incomplete or unstructured and generally of poor quality. For some programming languages (for example, JavaScript and Python), there is very little data and this is not enough to finetune neural networks. To solve these problems, we used large language models (LLM), generating synthetic data using them in two ways: generating comments from scratch and improving existing comments. For additional information about comment's enhancement see Appendix E.

Dataset Overview Table 1 presents the final statistical data of the final set, combining synthetic improved by the Miqu-70B model comments and generated from scratch by Qwen2.5-Coder-32B-Instruct ones with real comments from more than 150,000 Russian-language GitHub repositories of five programming languages: Python, Java, Go, C# and JavaScript. The total amount of data is 153,181 examples, of which 79,548 are improved, 65,914 are synthetic, and 7,719 are real comments.

[6]https://github.com/FSoft-AI4Code/CodeText-parser/tree/main

[7]https://github.com/nmd2k/docstring_parser

```
short description

long description

Args:
    name1 (type1): description1
    name2 (type2): description2

Returns:
    type: description

Raises:
    type: description
```

(a) Python Google docstring style

```
/**
 * short description
 *
 * long description
 *
 * @param name1 description1
 * @param name2 description2
 * @return description
 * @throws type description
*/
```

(b) JavaDoc comment style

```
/// <summary>
/// description
/// </summary>
///
/// <param name="name1">description1</param>
/// <param name="name2">description2</param>
///
/// <returns>description</returns>
///
/// <exception cref="type">description</exception>
```

(c) C# XML comment style

```
/**
 * short description
 *
 * long description
 *
 * @param {type1} name1 - description1
 * @param {type2} name2 - description2
 * @return {type} description
 * @throws {type} description
*/
```

(d) JSDOC comment style

```
// NameOfFunction description
```

(e) GoDoc comment style

Figure 1: Comparison of documentation styles in different programming languages

Prog. lang.	Enhanced	From scratch	Real
Python	14,625	10,078	359
Java	16,283	10,536	2,619
Go	7,278	20,339	232
C#	39,715	5,617	4,435
JavaScript	1,647	19,344	100
$\sum$	79,548	65,914	7,719

Table 1: Statistics of the collected Russian-language data on programming languages and methods of obtaining them. The table shows the amount of improved (modification of existing comments by the Miqu-70B model), generated from scratch (synthetic data from Qwen2.5-Coder-32B-Instruct) and real comments.

The uniqueness of the proposed dataset is determined by several factors (see Table 2). Firstly, this is the first large corpus with Russian-language documentation for functions. The only existing dataset with comments in Russian, MCoNaLa, is designed to solve a different problem - searching for a code snippet based on the user's intent and, therefore, is not suitable for generating structured comments in the *docstring* style. Secondly, our dataset was strictly checked for structure and completeness: all comments were modified to one of the formats used in the industry for each specific programming language. In other datasets, either there are no structured comments at all (MCoNaLa, CodeSearchNet), or they have not been filtered by structure (the Vault). Thirdly, as a result of the addition of synthetic data, the proposed set, unlike MCoNaLa, has a sufficient size to train large language models for all five selected programming languages.

4 Experimental Evaluation

We conducted experiments, where we first benchmark existing open-source code-specific LLMs of different size (Qwen2.5-Coder (0.5B - 7B) and DeepSeek-Coder (1.3B - 6.7B)), then finetune Qwen2.5-Coder (0.5B - 7B) on 7,500 comments, sampled from a synthetic part of our dataset and evaluate all models on our test set, 500 comments, sampled from real comments.

Feature	CSN	Vault	MCoNaLa	Our dataset
#Pairs «code-text»	6.5M	43K	341 - es, 210 - ja, 345 - ru	153K
Code format	Functions	Functions, classes, snippets	Code snippets	Functions
Text format	Unstr., 1-2 sent.	Mixed (unstr. and str. w/o filtration by structure)	Unstr., (1-2 sent.)	Str. complete (>5 sent.)
Progr. lang.	Go, Java, PHP, JavaScript, Python, Ruby	Java, JavaScript, Python, Ruby, Rust, Golang, C#, C++, C, PHP	Python, Java, JavaScript	Java, Python, C#, Go, JavaScript
Nat. lang.	en	en	ru, ja, es	ru
Data source	GitHub	The Stack	Stack Overflow	GitHub

Table 2: Comparison of the characteristics of the proposed dataset with existing analogues (CSN, Vault, MCoNaLa) by key parameters. The table shows the amount of data, the formats of code and text representation, the coverage of programming languages, linguistic features and data sources. The dataset we propose stands out with a strict focus on Russian-language structured comments on functions (153 thousand pairs), which contrasts with English-language counterparts operating with unstructured or mixed comments.

Evaluation with Textual Similarity Metrics We evaluated the models using standard natural language generation metrics, including chrF++ (Popović, 2017) and a modified BERTScore (Zhang et al.). Instead of the traditional BERT (Kenton and Toutanova, 2019), we employed E5-Mistral 7B (Wang et al., 2022, 2023a), which offers superior performance for Russian, outperforming BERT models. The results of evaluation are shown in Table 7.

Side-by-Side comparison The Side-by-Side comparison was performed with GitHub Copilot using LLM-as-a-judge method (the judge is GPT-4o-mini) (Zheng et al., 2023). Finetuning of models on our dataset leads to a great improvement in the quality of comment generation for all programming languages and model sizes, which is shown in Table 6. More details are presented in Appendix G.

Training and Results The additional information about training setup, hyperparameters, etc. is located in Appendix F. Finetuning on the proposed dataset significantly improves the quality of comment generation using the BERTScore metric for all model sizes and most languages. For chrF++, significant improvements are observed in small number of cases. The results confirm that the proposed approach is effective for adapting language models to the task of generating Russian-language comments, especially in terms of semantic correctness (BERTScore).

5 Conclusion

In this paper, we have developed a tool for filtering structured comments, collected a dataset of 153 thousand Russian-language code-comment pairs (real and synthetic data for 5 programming languages). We plan to expand the dataset by adding other programming languages, and develop and implement a quality criterion for structured code comments to automatically filter data and therefore improve the quality of the dataset.

6 Limitations

The study has several limitations, including a specific commenting style limitation, an imbalanced test dataset, and the assumption that code comments always contain useful information about code functionality, which is not always true. Additionally, some code comments from GitHub may be redundant, uninformative, or contain errors, negatively impacting the dataset's quality.

7 Acknowledgement

This research was supported by the Ministry of Economic Development of the Russian Federation (IGK 000000C313925P4C0002), agreement No139-15-2025-010. The authors express their sincere gratitude to the Ministry for the essential funding that enabled the pursuit of this work.

References

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search.

Armand Joulin, Edouard Grave, Piotr Bojanowski, Matthijs Douze, Hérve Jégou, and Tomas Mikolov. 2016. Fasttext.zip: Compressing text classification models. *arXiv preprint arXiv:1612.03651*.

Armand Joulin, Édouard Grave, Piotr Bojanowski, and Tomáš Mikolov. 2017. Bag of tricks for efficient text classification. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 2, Short Papers*, pages 427–431.

Jacob Devlin Ming-Wei Chang Kenton and Lee Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of naacL-HLT*, volume 1, page 2. Minneapolis, Minnesota.

Denis Kocetkov, Raymond Li, Loubna Allal, Jia Li, Chenghao Mou, Carlos Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro Werra, and Harm Vries. 2022. The stack: 3 tb of permissively licensed source code.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, MING GONG, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie LIU. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, volume 1.

Dung Nguyen, Le Nam, Anh Dau, Anh Nguyen, Khanh Nghiem, Jin Guo, and Nghi Bui. 2023. The vault: A comprehensive multilingual dataset for advancing code understanding and generation. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 4763–4788, Singapore. Association for Computational Linguistics.

Maja Popović. 2017. chrf++: words helping character n-grams. In *Proceedings of the second conference on machine translation*, pages 612–618.

Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 3505–3506.

Liang Wang, Nan Yang, Xiaolong Huang, Binxing Jiao, Linjun Yang, Daxin Jiang, Rangan Majumder, and Furu Wei. 2022. Text embeddings by weakly-supervised contrastive pre-training. *arXiv preprint arXiv:2212.03533*.

Liang Wang, Nan Yang, Xiaolong Huang, Linjun Yang, Rangan Majumder, and Furu Wei. 2023a. Improving text embeddings with large language models. *arXiv preprint arXiv:2401.00368*.

Zhiruo Wang, Grace Cuenca, Shuyan Zhou, Frank F. Xu, and Graham Neubig. 2023b. MCoNaLa: A benchmark for code generation from multiple natural languages. In *Findings of the Association for Computational Linguistics: EACL 2023*, pages 265–273, Dubrovnik, Croatia. Association for Computational Linguistics.

Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q Weinberger, and Yoav Artzi. Bertscore: Evaluating text generation with bert. In *International Conference on Learning Representations*.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. 2023. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems*, 36:46595–46623.

A Comment Extraction

To extract comments, we used the *function_parser*[8] tool for Python, Java, and Go. For JavaScript and C#, we employed *Code-Text*. The GitHub data collection process consisted of several steps. First, code snippets from Python and JavaScript libraries with very few non-English comments were excluded. The formatting of comments in Java, JavaScript, and C# was then standardized. In C#, XML tags such as <summary> were corrected. For Java and JavaScript, redundant whitespaces, line

[8]https://github.com/ncoop57/function_parser

breaks in block comments (delimited by /** and */), and HTML tags were removed. Next, automatically generated comments in C# and JavaScript were filtered out. Duplicate comments in the function and docstring columns were eliminated, along with duplicates based on function and docstring independently. The language of each comment was then identified using Lingua [9]. More information about language identification methods that we used is in Appendix B. If Lingua failed to determine the language, the corresponding comments were excluded from the dataset. To improve language identification accuracy, Lingua was provided with short descriptions of comments, ensuring tags and identifier names that could degrade identification quality were removed. This process was applied to all programming languages except Go, which has a relatively simple comment structure.

The final dataset, after filtering, is summarized in Table 3. The results show that JavaScript and Go are characterized by a similar trend: a high proportion of commented repositories (70.8% and 55.9%) and functions (70.2% and 25.8%) are combined with a low percentage of Russian-language comments (24.0% and 16.4%), which may indicate the predominance of English-language documentation in their ecosystems. On the contrary, Python and C# show an increased proportion of Russian—language comments (49.2% and 36.4%), which is probably due to regional development practices - the active participation of Russian-speaking communities in projects in these languages, where comments are often written in their native language for the local context.

B Language Identification

We applied two language identification methods to determine the language of the comments: FastText (Joulin et al., 2017, 2016) and Lingua. FastText uses a bag-of-n-grams approach to capture partial word order information, enabling efficient processing of large datasets on consumer hardware. Its pretrained models can classify text into one of 217 supported languages with high speed and efficiency. Lingua, on the other hand, employs a probabilistic n-gram model combined with rule-based heuristics, focusing on achieving high detection accuracy across 75 supported languages. While FastText offers broad language coverage and high efficiency, it demonstrated high precision but low recall for identifying Russian comments, frequently misclassifying them as less popular languages. Lingua, although slower and more memory-intensive, excels at handling short text and mixed-language inputs, which are common in code comments where natural language often intermixes with programming-specific syntax (e.g., tags and identifier names). Lingua's robustness in these scenarios makes it a preferable choice for detecting natural language within code comments.

C Comment Structure

The examples of comment structure for five selected programming languages are shown in Figure 1. Notably, Python's GoogleDoc and JavaScript's JSDoc are the only styles among the selected ones that require explicit descriptions of parameter types and return types, reflecting the dynamically-typed nature of these languages. JSDoc shares stylistic similarities with JavaDoc, emphasizing structured documentation. By contrast, C# utilizes XML for comment formatting, providing a more tag-based approach. GoDoc stands apart with its flexible and descriptive style, as it imposes no strict format requirements, allowing developers to use a nearly free-form commentary approach.

D Filtration by Structure

For filtration-by-structure stage, we utilized the fork of *docstring_parser* library [10] and *javalang* [11] tools to extract information about comment structure and *Code-Text* to gather information about code structure. We also added missing types in Python comments where possible using *Code-Text*. The dataset's collection showed significant differences in structured comments' availability and completeness across programming languages, as summarized in Table 4. The results demonstrate an inverse relationship between the complexity of the commenting standard and the proportion of complete structured comments. Go, with minimal requirements (only the function name at the beginning of the comment), shows the maximum percentage of full comments (56.4%, 10,880). On the contrary, Python and JavaScript, where standards require specifying types and complex annotations, have an extremely low proportion of complete comments (1.5% and 1.4%), with unstructured ones dominating (94,968 and 14,091). Java

[9] `https://github.com/pemistahl/lingua-py`

[10] `https://github.com/rr-/docstring_parser`

[11] `https://github.com/c2nes/javalang`

Programming language	#Repositories			#Functions			#Comments		
	With comments	Total	%	With comments	Total	%	in Russian	Total	% in Russian
Python	18,535	64,440	28.8%	305,187	1,627,726	18.7%	150,255	305,187	49.2%
Java	13,525	42,271	32.0%	409,506	2,684,650	15.3%	98,622	409,506	24.1%
Go	2,592	4,639	55.9%	117,691	456,347	25.8%	19,276	117,691	16.4%
C#	8,858	26,329	33.6%	291,142	596,905	48.8%	106,058	291,142	36.4%
JavaScript	15,073	21,291	70.8%	129,767	184,871	70.2%	31,084	129,767	24.0%

Table 3: Statistics on data collection from GitHub, including analysis of repositories, functions, and comments on programming languages, grouped into three categories: **repositories** (the total number of repositories for each programming language, the number of at least one comment, and the percentage of the latter), **functions** (the total number of functions, the number of functions with comments and their relative proportion) and **comments** (the total number of comments, the number of Russian-language comments and their percentage).

and C++ with moderately complex standards occupy an intermediate position: 29.8% and 22.7% of full comments, respectively, but a significant number of unstructured (48,347 and 30,188). The table confirms that the simpler the syntax of a structured comment, the higher the proportion of its compliance. The extremely high Go score is explained by the simplified standard, and the low Python/JavaScript values are due to the excessive complexity of the requirements, which leads to a preference for unstructured comments.

E Enhancement of Comments via LLM

The final dataset includes only those data with the length of both the code and the comment ranging from 250 to $1,000$ characters. Very short comments and functions were excluded, as the goal was to create a dataset with detailed and comprehensive documentation. Very long comments or features are outliers and therefore were not considered. Comments were generated from scratch using the Qwen2.5-Coder-32B-Instruct model for functions without comments (see Table 3) and for functions, which comments were not successfully enhanced. To improve the dataset, the MIQU 70B [12] model was used, which was further trained in Russian. The goal of the improvement is to generate a complete and detailed comment of the best quality based on the function and the existing comment on it. An example is illustrated in figure 2. System and user prompts used for mentioned two types of synthetic data collection are placed in Appendix, see 3, 4, 5 and 6, prompts for generation from scratch are in English, while the ones for enhancement are in Russian, as finetuned MIQU 70B works better with Russian prompts. Candidates for improvement were selected from all the structuredness groups that were not included in the dataset in the “real” group. Comment is considered improved if it has become complete as a result of the improvement. Table 5 shows statistics on improving the dataset. Go stands out for the maximum efficiency of improvements (avg = 84.3%), especially for complete comments (91.5%), which is explained by a simple commenting standard, where it is enough to specify the function name. Python and JavaScript show the lowest averages (31.9% and 33.5%), which is due to the complexity of their standards, which require specifying data types, which makes automatic modification difficult. C# and Java occupy an intermediate position: C# shows a high average percentage of improvements (80.1%) with a peak in the full comments category (92.4%), while Java shows moderate results (avg = 48.2%).

[12]https://huggingface.co/miqudev/miqu-1-70b

F Training and Results

The models were trained for 5 epochs with a context length of 2000, a learning rate of 1e-4, and a cosine scheduler with a weight decay of 0.1 and a warmup ratio of 0.01. We used LORA (Hu et al., 2021) adapters with a rank of 8, alpha of 16, and a dropout rate of 0.05 for finetuning. From the synthetic part of the dataset, we sampled 1,500 examples for each programming language, resulting in 7,500 examples. For calculating metrics on real data, we sampled 100 examples for each programming language. The comparison is made with the base models to determine the extent to which training on our synthetic dataset improves the quality. Notably, with a batch size of 1, the model takes approximately 20 hours to train on 5 programming languages using DeepSpeed Zero2 (Rasley et al., 2020) on a single A100 GPU. The results are shown

Programming language	Structured			Non-structured
	% complete out of all Russian	Complete	Incomplete	
Python	1.5%	2,176	30,115	94,968
Java	29.8%	29,367	12,221	48,347
Go	56.4%	10,880	-	8,396
C#	22.7%	24,017	41,898	30,188
JavaScript	1.4%	431	1,484	14,091

Table 4: The structure of Russian-language comments on programming languages. For each language, the following are indicated: the percentage of complete structured comments out of the total number of Russian-language comments (% of the total number), the absolute values of complete and incomplete structured comments, as well as the number of unstructured ones. In Go, the dash in the “Incomplete” column is due to a feature of the commenting standard: comments are considered complete if they begin with the function name, which excludes the “incomplete” category.

Programming language		Non-structured	Incomplete	Complete	
Python	#Enhanced comments	10 775	3 455	395	$\sum$ = 14 625
	% out of the original quantity	24.2%	23.2%	48.1%	avg = 31.9%
Java	#Enhanced comments	7 066	3 810	5 407	$\sum$ = 16 283
	% out of the original quantity	32.0%	57.6%	55.1%	avg = 48.2%
Go	#Enhanced comments	3 018	-	4 260	$\sum$ = 7 278
	% out of the original quantity	77.1%	-	91.5%	avg = 84.3%
C#	#Enhanced comments	12 467	18 148	9 100	$\sum$ = 39 715
	% % out of the original quantity	74.8%	73.1%	92.4%	avg = 80.1%
JS	#Enhanced comments	1 386	164	97	$\sum$ = 1 647
	% % out of the original quantity	20.4%	20.4%	59.5%	avg = 33.5%

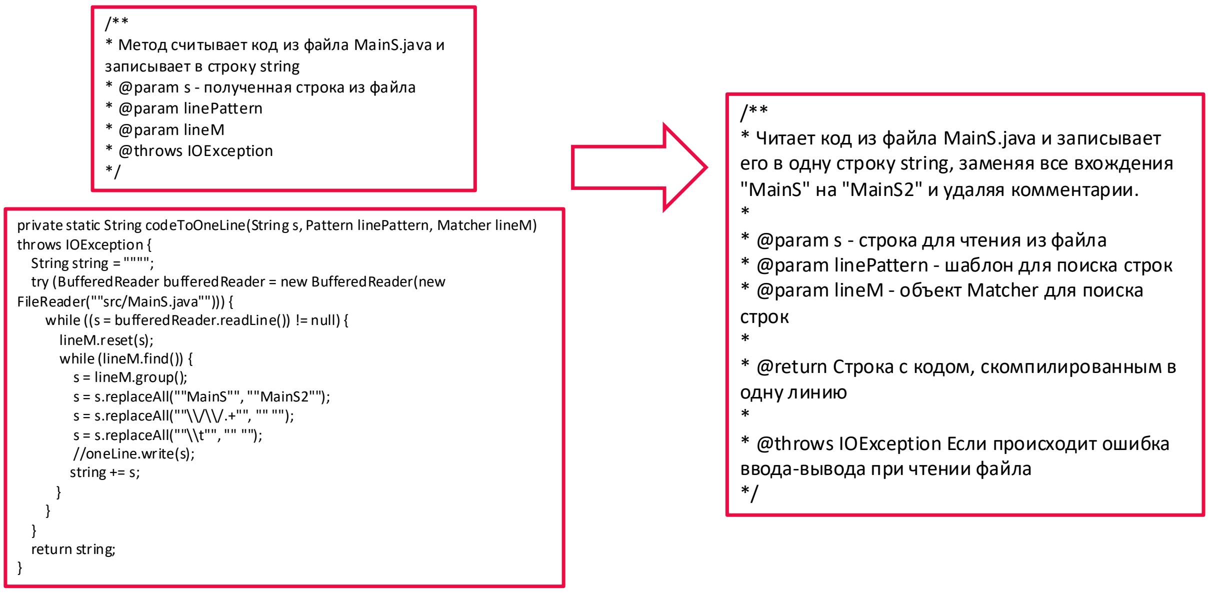
Table 5: Statistics on the improvement of Russian-language comments on programming languages, divided into categories: unstructured, incomplete and complete structured comments. For each language, the absolute number of improved comments, the percentage of improvements relative to the initial number in the category (from the Table 4), the total number of improvements ($\sum$) and the average percentage of improvements (avg) are indicated. The dash in the category of incomplete comments for Go reflects their absence in the source data due to the simplified standard for documenting functions.

in Table 7.

G Side-by-side Comparison

We adopt the LLM-as-a-judge paradigm (Zheng et al., 2023), leveraging GPT-4’s RLHF-aligned reasoning for automated pairwise comparisons, see Table 6. This approach replaces costly expert labeling while maintaining 80% human judgment consistency and providing interpretable rationales. To mitigate positional bias, responses are evaluated twice with reversed order, recording victories only for consistent outcomes. Our implementation introduces two tie types: «win» (both responses adequate) and «lose» (both inadequate), refining outcome granularity for semantically similar comments.

The greatest progress is observed in compact models (0.5B-3B parameters), which indirectly confirms the hypothesis of high data relevance: smaller architectures are more dependent on the quality of training examples, and their visible performance growth indicates a successful dataset selection that compensates for the lack of parameters.



```
/**
* Метод считывает код из файла MainS.java и
записывает в строку string
* @param s - полученная строка из файла
* @param linePattern
* @param lineM
* @throws IOException
*/
```

```
private static String codeToOneLine(String s, Pattern linePattern, Matcher lineM)
throws IOException {
   String string = """";
   try (BufferedReader bufferedReader = new BufferedReader(new
FileReader(""src/MainS.java""))) {
      while ((s = bufferedReader.readLine()) != null) {
         lineM.reset(s);
         while (lineM.find()) {
            s = lineM.group();
            s = s.replaceAll(""MainS"", ""MainS2"");
            s = s.replaceAll(""\\/\\/.+"", "" "");
            s = s.replaceAll(""\\t"", "" "");
            //oneLine.write(s);
           string += s;
         }
      }
   }
   return string;
}
```

```
/**
* Читает код из файла MainS.java и записывает
его в одну строку string, заменяя все вхождения
"MainS" на "MainS2" и удаляя комментарии.
*
* @param s - строка для чтения из файла
* @param linePattern - шаблон для поиска строк
* @param lineM - объект Matcher для поиска
строк
*
* @return Строка с кодом, скомпилированным в
одну линию
*
* @throws IOException Если происходит ошибка
ввода-вывода при чтении файла
*/
```

Figure 2: An example of improving a comment. On the left is a function and a comment on it before improvement, which (1) fails to explain the method's purpose (converting code into a single line with modifications), (2) contains an incorrect description of parameter "s" (presenting it as the result when it's actually a buffer), (3) completely ignores the return value, (4) omits key operations: replacing "MainS" → "MainS2", removing comments (//...), and deleting tabulations. The comment after the improvement is devoid of these shortcomings.

You are an AI programming assistant. Follow the user's requirements carefully & to the letter.

Figure 3: System prompt for generation from scratch

Please provide documentation comments (Docstring, GoDoc, JavaDoc, JSDoc, XML docs, etc., depending on language) to this function. На русском языке, пожалуйста.

Figure 4: User prompt for generation from scratch

Вы опытный программист, который вышел на пенсию и сейчас помогает советом своим коллегам. У вас много свободного времени, поэтому вы читаете всю новейшую литературу в данной области, а также готовы прийти на помощь любому попросившему в любой момент. Вы в совершенстве знаете языки программирования Java, Python, Go, C#, JavaScript и их стили документации - JavaDoc (Java), JSDoc(JavaScript), GoDoc(Go), XML (C#) .

Вы терпеливы, умеете объяснять в деталях каждое конкретное решение, но при этом задание выполняете максимально лаконично. Вы всегда предельно вежливы и отзывчивы.

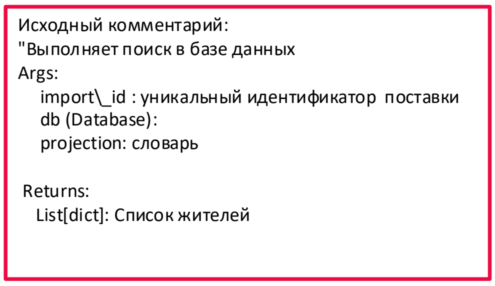
Ваша главная задача - помогать составлять комментарии к коду на тех языках программирования, которые вы знаете. Комментарии должны быть только на {right} языке, если Вы сгенерируете комментарий на {wrong}, то Вас оштрафуют на 1000000 долларов.

Ваш ответ должен состоять только из комментария к функции в стиле, определённом для конкретного языка программирования

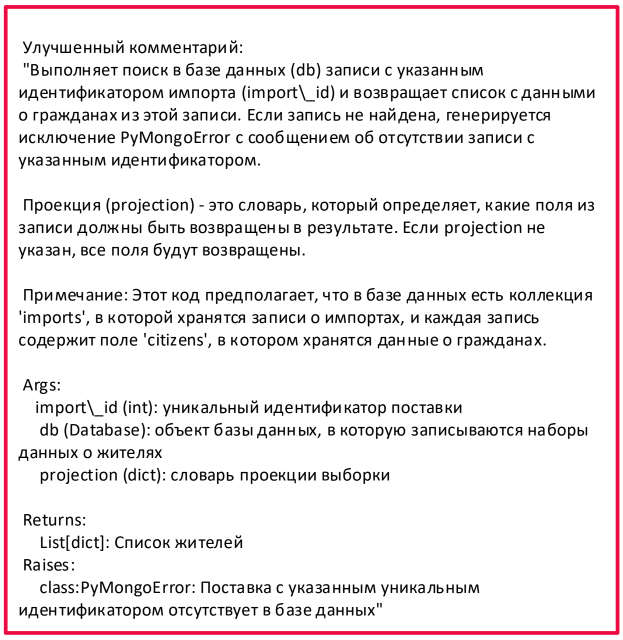
Не забывайте оборачивать текст в комментариях к коду на C# в тэг <summary>.

Помните, что для Go общепринятый стандарт не устанавливает жесткие ограничения на структуру комментария, достаточно, чтобы первое слово в комментарии было названием комментируемой функции. Нужно, чтобы комментарии для Go были максимально подробными, и по смыслу включали в себя то, что и в других языках (описание параметров и их типы, тип и описание возвращаемого значения, а также тип исключения с описание ситуации, когда оно выбрасывается).

Figure 5: System prompt for the enhancement of existing comments


Исходный комментарий:
"Выполняет поиск в базе данных
Args:
    import_id : уникальный идентификатор поставки
    db (Database):
    projection: словарь

 Returns:
   List[dict]: Список жителей


 Улучшенный комментарий:
 "Выполняет поиск в базе данных (db) записи с указанным идентификатором импорта (import_id) и возвращает список с данными о гражданах из этой записи. Если запись не найдена, генерируется исключение PyMongoError с сообщением об отсутствии записи с указанным идентификатором.

 Проекция (projection) - это словарь, который определяет, какие поля из записи должны быть возвращены в результате. Если projection не указан, все поля будут возвращены.

 Примечание: Этот код предполагает, что в базе данных есть коллекция 'imports', в которой хранятся записи о импортах, и каждая запись содержит поле 'citizens', в котором хранятся данные о гражданах.

 Args:
    import_id (int): уникальный идентификатор поставки
    db (Database): объект базы данных, в которую записываются наборы данных о жителях
    projection (dict): словарь проекции выборки

 Returns:
    List[dict]: Список жителей
 Raises:
    class:PyMongoError: Поставка с указанным уникальным идентификатором отсутствует в базе данных"

Figure 6: User prompt for the enhancement of existing comments

Experiment type	Python	Java	Go	C#	JavaScript
			Qwen2.5-Coder-7B-Instruct		
w/o finetuning	**48.0**/2.0/16.5/33.5	**65.5**/6.0/1.0/27.5	**43.5**/3.5/6.0/47.0	**22.0**/2.0/3.0/74.0	**44.0**/4.5/3.0/48.5
w finetuning	**45.0**/6.5/19.0/29.5	**85.0**/4.0/0.5/10.5	**61.0**/5.5/5.0/28.5	**81.0**/3.5/2.0/13.5	**71.0**/2.0/0.0/27.0
			Qwen2.5-Coder-3B-Instruct		
w/o finetuning	**7.0**/0.0/16.5/76.5	**24.0**/0.5/2.5/73.0	**7.0**/0.5/4.5/88.0	**7.0**/0.0/4.5/88.5	**19.5**/0.5/5.0/75.0
w finetuning	**41.5**/4.5/21.0/33.0	**81.5**/6.0/0.0/12.5	**58.0**/3.5/4.5/34.0	**82.0**/4.0/2.0/12.0	**65.5**/5.0/0.5/29.0
			Qwen2.5-Coder-1.5B-Instruct		
w/o finetuning	**18.5**/0.5/16.5/64.5	**20.0**/1.0/3.5/75.5	**9.5**/0.0/8.5/82.0	**7.0**/0.5/2.0/90.5	**13.5**/0.0/3.5/83.0
w finetuning	**38.0**/2.5/26.0/33.5	**78.0**/4.0/1.5/16.5	**58.0**/4.5/6.5/31.0	**73.0**/4.5/3.5/19.0	**58.5**/4.5/1.0/36.0
			Qwen2.5-Coder-0.5B-Instruct		
w/o finetuning	**36.0**/2.0/25.0/37.0/	**24.5**/0.5/4.5/70.5	**12.5**/0.0/13.5/74.0	**5.5**/0.5/5.0/89.0	**8.5**/0.0/4.0/87.5
w finetuning	**18.0**/1.0/22.0/59.0	**60.0**/3.5/2.0/34.5	**31.5**/2.0/5.0/61.5	**53.5**/2.5/4.0/40.0	**41.0**/1.5/2.0/55.5

Table 6: The results of the Side-by-side evaluation with the GPT-4o-mini judge. The estimates are presented as: Model VS Copilot, win/win_tie/lose_tie/lose, which corresponds to the estimates of 10/11/00/01. The answers were evaluated twice with a change in their order to solve the problem of positional bias.

Model	Python		Java		Go		C#		JavaScript	
	BERTScore	chrF++	BERTScore	chrF++	BERTScore	chrF++	BERTScore	chrF++	BERTScore	chrF++
Baselines										
DeepSeek-Coder 1.3B	0.837 ±0.041	18.3 ±9.8	0.827 ±0.040	19.2 ±7.2	0.811 ±0.042	10.4 ±4.5	0.812 ±0.044	18.4 ±16.9	0.839 ±0.038	24.7 ±8.7
DeepSeek-Coder 6.7B	0.878 ±0.043	34.1 ±10.5	0.873 ±0.044	36.9 ±14.2	0.838 ±0.047	21.0 ±11.1	0.844 ±0.052	36.3 ±18.2	0.876 ±0.033	38.4 ±10.9
Qwen2.5-Coder 0.5B	0.863 ±0.052	26.6 ±9.8	0.839 ±0.056	20.7 ±9.3	0.816 ±0.052	10.9 ±5.6	0.815 ±0.052	14.1 ±8.5	0.799 ±0.035	9.6 ±6.1
Qwen2.5-Coder 1.5B	0.841 ±0.045	22.8 ±10.8	0.838 ±0.045	21.2 ±10.5	0.815 ±0.039	11.5 ±5.0	0.821 ±0.051	31.5 ±14.9	0.841 ±0.035	23.8 ±7.9
Qwen2.5-Coder 3B	0.784 ±0.061	14.2 ±8.4	0.829 ±0.039	17.2 ±6.0	0.819 ±0.041	11.0 ±4.4	0.817 ±0.046	25.7 ±15.5	0.841 ±0.033	23.7 ±6.2
Qwen2.5-Coder 7B	0.880 ±0.040	34.3 ±7.7	0.873 ±0.039	35.0 ±9.8	0.854 ±0.039	23.5 ±9.1	0.847 ±0.037	24.3 ±12.2	0.872 ±0.031	33.5 ±7.9
Finetuned Models										
Qwen2.5-Coder 0.5B	0.873 ±0.042	**35.3 ±9.0**	**0.872 ±0.040**	39.7 ±9.8	**0.859 ±0.038**	28.7 ±6.8	**0.849 ±0.041**	44.4 ±10.2	**0.871 ±0.035**	40.3 ±0.03
Qwen2.5-Coder 1.5B	**0.877 ±0.040**	34.4 ±7.5	**0.880 ±0.036**	41.6 ±8.8	**0.863 ±0.035**	32.1 ±6.3	**0.857 ±0.038**	45.7 ±9.3	**0.877 ±0.031**	40.3 ±0.03
Qwen2.5-Coder 3B	**0.880 ±0.040**	34.9 ±7.5	**0.881 ±0.035**	40.6 ±8.3	**0.864 ±0.035**	32.5 ±6.2	**0.859 ±0.037**	46.4 ±9.7	**0.878 ±0.031**	41.3 ±8.5
Qwen2.5-Coder 7B	0.878 ±0.039	35.5 ±7.3	0.882 ±0.036	**42.0 ±8.9**	**0.867 ±0.035**	32.9 ±6.2	**0.859 ±0.034**	45.9 ±9.5	0.879 ±0.032	**41.4 ±7.6**

Table 7: Comparison of base and finetuned models using BERTScore and chrF++ metrics with statistical significance testing (Mann-Whitney criterion). Statistically significant improvements ($p < 0.05$) are highlighted in **bold** when comparing the finetuned model with the corresponding sized base version. The values are presented as the average ± standard deviation.