RESEARCH-ARTICLE

Less Is More: DocString Compression in Code Generation

GUANG YANG, Nanjing University of Aeronautics and Astronautics, Nanjing, Jiangsu, China

YU ZHOU, Nanjing University of Aeronautics and Astronautics, Nanjing, Jiangsu, China

WEI CHENG, Nanjing University of Aeronautics and Astronautics, Nanjing, Jiangsu, China

XIANGYU ZHANG, Nanjing University of Aeronautics and Astronautics, Nanjing, Jiangsu, China

XIANG CHEN, Nantong University, Nantong, Jiangsu, China

TERRY YUE ZHUO, Monash University, Melbourne, VIC, Australia

View all

Published: 21 January 2026
Online AM: 14 May 2025
Accepted: 06 May 2025
Revised: 24 April 2025
Received: 30 October 2024

Less Is More: DocString Compression in Code Generation

GUANG YANG, College of Computer Science and Technology/College of Artificial Intelligence/College of Software, Nanjing University of Aeronautics and Astronautics, Nanjing, China
YU ZHOU, WEI CHENG, and XIANGYU ZHANG, Nanjing University of Aeronautics and Astronautics, Nanjing, China
XIANG CHEN, Nantong University, Nantong, China
TERRY YUE ZHUO, Monash University, Clayton, Australia
KE LIU, National University of Defense Technology, Changsha, China
XIN ZHOU and DAVID LO, Singapore Management University, Singapore, Singapore
TAOLUE CHEN, Birkbeck University of London, London, United Kingdom of Great Britain and Northern Ireland

The widespread use of Large Language Models (LLMs) in software engineering has intensified the need for improved model and resource efficiency. In particular, for neural code generation, LLMs are used to translate function/method signature and DocString to executable code. DocStrings, which capture user requirements for the code and are typically used as the prompt for LLMs, often contain redundant information. Recent advancements in prompt compression have shown promising results in Natural Language Processing (NLP), but their applicability to code generation remains uncertain. Our empirical study shows that the state-of-the-art prompt compression methods achieve only about 10% reduction, as further reductions would cause significant performance degradation. In our study, we propose a novel compression method, ShortenDoc, dedicated to DocString compression for code generation. Our experiments on six code generation datasets, five open source LLMs (1B to 10B parameters), and one closed-source LLM GPT-4o confirm that ShortenDoc achieves 25–40% compression while preserving the quality of generated code, outperforming other baseline

This research/project is supported by the National Natural Science Foundation of China (No. 62372232), the Short-term Visiting Program of Nanjing University of Aeronautics and Astronautics for Ph.D. Students Abroad (No. 240602DF16), High Performance Computing Platform of Nanjing University of Aeronautics and Astronautics, and the Collaborative Innovation Center of Novel Software Technology and Industrialization. T. Chen is partially supported by an oversea grant from the State Key Laboratory of Novel Software Technology, Nanjing University (KFKT2022A03, KFKT2023A04). This research/project is supported by the A*STAR under its 2nd CSIRO and A*STAR: Research-Industry (2+2) Partnership Program (Award R24I5IR047). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the A*STAR.
Authors' Contact Information: Guang Yang, College of Computer Science and Technology/College of Artificial Intelligence/College of Software, Nanjing University of Aeronautics and Astronautics, Nanjing, China; e-mail: novelyg@outlook.com; Yu Zhou (corresponding author), Nanjing University of Aeronautics and Astronautics, Nanjing, China; e-mail: zhouyu@nuaa.edu.cn; Wei Cheng, Nanjing University of Aeronautics and Astronautics, Nanjing, China; e-mail: chengweii@nuaa.edu.cn; Xiangyu Zhang, Nanjing University of Aeronautics and Astronautics, Nanjing, China; e-mail: zhangx1angyu@nuaa.edu.cn; Xiang Chen, Nantong University, Nantong, China; e-mail: xchencs@ntu.edu.cn; Terry Yue Zhuo, Monash University, Clayton, Australia; e-mail: terry.zhuo@monash.edu; Ke Liu, National University of Defense Technology, Changsha, China; e-mail: liuke23@nudt.edu.cn; Xin Zhou, Singapore Management University, Singapore, Singapore; e-mail: xinzhou.2020@phdcs.smu.edu.sg; David Lo, Singapore Management University, Singapore, Singapore; e-mail: davidlo@smu.edu.sg; Taolue Chen (corresponding author), Birkbeck University of London, London, United Kingdom of Great Britain and Northern Ireland; e-mail: t.chen@bbk.ac.uk.

ACM 1557-7392/2026/1-ART41
https://doi.org/10.1145/3735636

methods at similar compression levels. The benefit of this method is to improve efficiency and reduce the token processing cost while maintaining the quality of the generated code, especially when calling third-party APIs.

CCS Concepts: • **Software and its engineering**; • **Computing methodologies** → **Artificial intelligence**;

Additional Key Words and Phrases: DocString Compression, Code Generation, Large Language Model

ACM Reference format:
Guang Yang, Yu Zhou, Wei Cheng, Xiangyu Zhang, Xiang Chen, Terry Yue Zhuo, Ke Liu, Xin Zhou, David Lo, and Taolue Chen. 2026. Less Is More: DocString Compression in Code Generation. *ACM Trans. Softw. Eng. Methodol.* 35, 2, Article 41 (January 2026), 31 pages.
https://doi.org/10.1145/3735636

1 Introduction

Large Language Models (LLMs) have emerged as a key tool in today's software development, owning to their powerful language understanding and generation capabilities [30, 41]. In particular, they have shown great potential in a variety of SE tasks such as code generation [23], automated testing [7], and documentation [11]. However, with the wide deployment of LLMs, it has become a pressing challenge to improve the inference efficiency and to reduce the ever increasing demand of computational resources [33, 42, 55]. This challenge manifests in two aspects: (1) the computational overhead in terms of **Floating Point Operations (FLOPs)** during model inference, and (2) the financial cost associated with API calls to third-party LLMs for code generation.

DocString (documentation string), a string literal embedded in source code to document specific features or functionalities, serves as a critical component in software development. It significantly enhances code readability, maintainability, and usability. Typically positioned at the beginning of a function or method, DocString plays a pivotal role in the code generation process by articulating the code's functionality, parameters, return values, and potential exceptions that may arise [8]. Listing 1 presents an example to illustrate how DocString (Lines 2–14) elucidates the purpose of the function, its parameters, the type of return value it yields, the exceptions it might trigger, and offers a practical example showing how to invoke the function.

Motivation. In the realm of code generation leveraging LLMs, DocStrings are typically used as prompts. They guide the model to generate code that aligns with the document specific features or functionalities [26]. However, existing DocString varies significantly in quality [37]. Some DocString may be overly lengthy and contain a large amount of redundancy, which may prevent the model from accurately understanding user requirements. Although these redundant DocString do not exceed the length of the maximum context that LLM can now handle, they may also increase the computational cost of the model inference, thus reduce their efficiency. For instance, the computational cost of the model inference is usually proportional to the length of the input prompt. The longer the DocString, the higher the computational cost of model inference. According to OpenAI founder Sam Altman, GPT-4o processes approximately 200 billion tokens daily,[1] while Baidu CEO Robin Li revealed (at Baidu World 2024) that ERNIE model handles over 1.5 billion calls per day.[2] At such massive processing scales, even a modest 25–40% compression of user inputs could significantly reduce overall computational costs and energy consumption in the long run.

In addition, when users generate code by invoking APIs from third-party LLMs, shorter prompt can greatly reduce their financial cost. Therefore, optimizing the quality of DocString is one of the key factors for improving the efficiency of code generation and reducing costs. For instance, the

[1] https://x.com/sama/status/1815437745550172617.
[2] https://ir.baidu.com/news-releases/news-release-details/baidu-announces-third-quarter-2024-results/.

```python
def add_numbers(a, b):
    """
    Adds two numbers and returns the result.
        Parameters:
    a (int or float): The first number to add.
    b (int or float): The second number to add.
        Returns:
    int or float: The sum of a and b.
        Raises:
    TypeError: If 'a' or 'b' is not a number.
        Example:
    >>> add_numbers(2, 3)
    5
    """
    if not (isinstance(a, (int, float)) and isinstance(b, (int, float))):
        raise TypeError("Both arguments must be numbers")
    return a + b
```

Listing 1. The Example of DocString in Code Generation

costs of LLM API are associated with input tokens (i.e., prefilling) and output tokens (i.e., decoding), where input tokens are usually 1–5 times cheaper than output tokens (varying across different LLM providers). For example, Meta's Llama 3.1 405B model charges at the same rate for both input and output tokens[3]; OpenAI's models price input tokens at 1/4 of output tokens[4]; Anthropic's models set input tokens at 1/5 of output token costs.[5] Assuming one usage consumes 512 input tokens and 512 output tokens on average, with a 50% input token reduction, the end-to-end cost saving is $(512 \times 1 \times 50\%)/(512 \times 1 + 512 \times 4) = 10\%$ according to OpenAI's pricing model, and 25% according to Meta's pricing model. While the cost reduction for a single API call might seem modest due to the pricing difference between input and output tokens across LLM providers, the cumulative economic benefits become substantial in large-scale deployment scenarios or when synthesizing large-scale code generation datasets using LLMs.

Moreover, DocString compression can be complementary to other efficiency-oriented techniques in code generation, such as efficient decoding strategies [43, 44] and optimized programming language grammar [45]. Meanwhile, the insights gained from DocString compression in zero-shot code generation scenarios can also provide theoretical foundations for more complex retrieval-augmented generation based code generation tasks.

Naturally, one may apply existing prompt compression techniques (e.g., Selective_Context [21] and LLMLingua [15]) from **Natural Language Processing (NLP)** to DocString. It turns out that there are several challenges:

(1) *Compression Efficacy*: These compression methods show limited effectiveness in compressing DocStrings. As evidenced by our experimental results in Section 3 (Table 1), increasing the compression rate beyond 10% may incur a significant decrease in the quality of the generated code. We posit that this is largely because they fail to extract the code-related semantic information in DocStrings, leading to significant information loss.

(2) *Inflexibility*: Existing methods require manual setting of the compression ratio, making it difficult to adapt to various code generation scenarios. Developers must adjust compression

[3]https://llama3-1.com/price/.
[4]https://openai.com/api/pricing/.
[5]https://www.anthropic.com/pricing.

Table 1. Pass@1 Comparison of Existing Compression Methods during Different Ratios

Model	Dataset		Selective_Context (%)				LLMLingua2 (%)				
		0	10	20	30	40	10	20	30	40	100
DS-1.3B	HumanEval	63.42	54.88	50.00	44.51	51.22	53.05	51.22	42.68	37.20	21.95
	CodeHarmony	59.48	58.82	56.21	54.90	52.29	58.82	58.82	54.25	50.33	39.22
	MBPP	36.40	36.60	35.40	35.60	34.20	35.60	36.80	34.20	31.80	25.00
	Subtle	53.00	53.00	44.00	45.00	40.00	50.00	46.00	44.00	32.00	15.00
	Creative	23.00	21.00	20.00	17.00	14.00	20.00	22.00	17.00	14.00	2.00
	BigCodeBench	6.10	4.10	2.00	0.00	2.00	4.10	4.10	2.00	2.00	0.70
	Avg.	40.23	38.07	34.60	32.84	32.29	36.93	36.49	32.36	27.89	17.20
DS-6.7B	HumanEval	71.95	70.12	64.63	61.59	57.32	73.17	62.20	54.27	46.34	25.61
	CodeHarmony	64.36	66.67	67.97	67.32	59.48	67.32	66.67	67.97	63.40	45.75
	MBPP	46.20	47.60	46.60	46.20	45.00	45.80	44.80	46.20	42.00	30.60
	Subtle	61.00	58.00	55.00	55.00	47.00	58.00	60.00	45.00	39.00	13.00
	Creative	34.00	34.00	31.00	31.00	23.00	33.00	37.00	31.00	27.00	4.00
	BigCodeBench	12.20	8.80	4.70	6.10	6.10	12.20	9.50	6.10	5.40	0.00
	Avg.	48.29	47.53	44.98	44.54	39.65	48.25	46.70	41.76	37.19	19.83
CQ-7.3B	HumanEval	77.44	76.22	73.78	70.12	67.07	82.32	73.78	65.24	57.32	31.10
	CodeHarmony	60.78	60.78	58.82	59.48	58.82	64.05	63.40	62.75	58.82	49.67
	MBPP	59.00	58.20	57.80	57.40	55.60	58.20	56.20	53.00	51.60	35.60
	Subtle	63.00	54.00	54.00	52.00	51.00	65.00	57.00	55.00	48.00	14.00
	Creative	38.00	32.00	36.00	33.00	23.00	34.00	33.00	32.00	25.00	6.00
	BigCodeBench	13.50	8.80	6.10	6.10	5.40	9.50	8.80	6.10	7.40	0.00
	Avg.	51.95	48.33	47.75	46.35	43.48	52.18	48.70	45.68	41.36	22.73
CG-9.4B	HumanEval	60.37	62.20	54.88	56.10	49.39	62.80	61.59	52.44	37.20	20.73
	CodeHarmony	64.71	59.48	60.78	60.78	58.17	61.44	63.40	57.52	57.52	39.22
	MBPP	46.60	45.80	43.40	44.00	42.40	45.20	45.60	42.80	40.80	29.60
	Subtle	64.00	57.00	53.00	52.00	46.00	66.00	59.00	51.00	40.00	15.00
	Creative	36.00	33.00	31.00	32.00	28.00	36.00	39.00	32.00	21.00	3.00
	BigCodeBench	14.20	4.10	4.70	8.10	6.10	9.50	7.40	8.80	4.70	0.00
	Avg.	47.65	43.60	41.29	42.16	38.34	46.82	46.00	40.76	33.54	17.93
LA-8.0B	HumanEval	57.32	58.54	54.88	53.05	47.56	56.71	54.27	45.12	35.37	20.73
	CodeHarmony	59.48	58.17	56.86	58.17	58.17	62.09	62.75	62.75	54.90	43.79
	MBPP	41.40	42.20	39.80	41.60	37.00	41.00	41.20	38.40	35.20	24.20
	Subtle	56.00	57.00	59.00	50.00	47.00	58.00	53.00	47.00	35.00	16.00
	Creative	34.00	29.00	30.00	24.00	21.00	38.00	34.00	28.00	25.00	2.00
	BigCodeBench	10.80	1.40	3.40	2.70	4.70	8.10	7.40	5.40	4.10	0.00
	Avg.	43.17	41.05	40.66	38.25	35.91	43.98	42.10	37.78	31.60	17.79

parameters based on specific situations, which requires specialized knowledge and is time-consuming. Moreover, it is hard to find the optimal balance of compression ratio and model efficacy (specifically Pass@1, which measures the percentage of tasks for which the model produces a correct output on its first attempt).

Our Work. We aim to develop prompt compression methods dedicated to DocStrings, optimizing the efficiency of LLMs in code generation tasks. We introduce ShortenDoc, a novel method that

dynamically adjusts DocString compression based on the importance of individual tokens. Instead of using a fixed compression ratio, ShortenDoc analyzes the significance of each token, ensuring that essential information is preserved while removing redundant content.

Initially, we decompose a given prompt into its signature and DocString components. The DocString undergoes pre-processing to enhance its quality based on empirical findings. Each token in the DocString is assigned an importance score, and tokens are ranked accordingly. We then construct a search space of tokens eligible for compression, considering pre-defined constraints to ensure minimal semantic distortion.

Tokens are iteratively compressed until the constraint is no longer satisfied. The final compressed DocString is obtained by integrating the selected tokens with the original sequence. This approach ensures that ShortenDoc balances efficiency with accuracy, minimizing computational overhead while maintaining the quality of the generated code across different programming languages and contexts.

We evaluate the performance of ShortenDoc across six datasets and six LLMs, demonstrating its superior ability to preserve essential information and reduce inference costs compared to baseline methods. To explore the generalization of ShortenDoc, we evaluate the performance of ShortenDoc across four more programming languages. Additionally, we conduct a human study to further assess the practical impact of DocString compression, evaluating both informativeness and comprehensibility from the perspective of human evaluation.

Our contributions can be summarized as follows.

- —We empirically show the feasibility and limitations of DocString compression for code generation tasks.
- —We design a novel method ShortenDoc, which is an adaptive compression method for compressing DocString and has better compression results compared to existing methods.
- —We delve into the insights gained from DocString compression techniques and give relevant insights.

Structure. The rest of the article is organized as follows. Section 2 provides preliminary knowledge related to our study. Section 3 confirms and analyzes the feasibility and limitations of existing DocString compression methods for code generation tasks. Section 4 describes the key components of ShortenDoc. Section 5 present the **Research Questions (RQs)** and the result analysis, which is followed by the discussion in Section 6. Section 7 reviews the related work. Section 8 concludes our study and outlines future directions.

To facilitate reproducibility, all source code and experimental data are released at https://github.com/NTDXYG/ShortenDoc.

2 Background

Code generation, in a nutshell, refers to automated generation of code from specifications and under certain constraints, which plays a pivotal role in software development [20]. In this context, signatures and DocStrings (in natural language) are typically used as inputs to the model and are converted into executable code.

Let $\mathcal{T}$ denote the set of signatures, $\mathcal{D}$ denote the set of DocStrings, and $\mathcal{Y}$ denote the set of executable code. In general, signatures define the name of the function/method, input parameters, return type, and possible side effects. DocStrings provide a natural language description of the function or method, including its purpose, behavior, parameter explanations, and return values. Typically, code generation can be formalized as a function $f : \mathcal{T} \times \mathcal{D} \rightarrow \mathcal{Y}$.

In practice, most of the current code generation models follow Transformer's decoder architecture [47]. The process can be broken down into several steps, including word embedding, layer transformations, and vocabulary mapping.

Embedding. The input signature $T \in \mathcal{T}$ and DocString $D \in \mathcal{D}$ are tokenized into a sequence of words $w_1, \ldots, w_n$. Each word is then converted into a vector representation through word embedding. Let $\mathcal{W}$ be the vocabulary of tokens and $\mathbf{E} \in \mathbb{R}^{|\mathcal{W}| \times d_{\text{embed}}}$ be the embedding matrix, where d_{embed} is the dimensionality of the embedding space. The word embedding function $\text{Emb} : \mathcal{W} \rightarrow \mathbb{R}^{d_{\text{embed}}}$ maps each token to its corresponding vector representation, viz., $\text{Emb}(w_i)$ is the embedding vector for the ith word.

Transformer. The embedded sequence is then fed into a stack of L transformer layers to produce a sequence of hidden states $\mathbf{H} = [\mathbf{h}_1, \mathbf{h}_2, \ldots, \mathbf{h}_L]$, where $\mathbf{h}_i$ is the hidden state vector for the ith layer output and each Transformer-Block computes the hidden state through Self-Attention and Feed-Forward, i.e.,

$$\mathbf{h}_{i+1} = \text{Transformer-Block}(\mathbf{h}_i) = \text{Self-Attention}\ (\mathbf{h}_i) + \text{Feed-Forward}(\mathbf{h}_i).$$

Probability. The final hidden state vector $\mathbf{h}_L$ of the last layer is mapped to a vocabulary space using a linear transformation, followed by a softmax function to obtain a probability distribution over the vocabulary: $\mathbf{z} = \mathbf{W}\mathbf{h}_L + \mathbf{b}$, where $\mathbf{W}$ is the weight matrix and $\mathbf{b}$ is the bias vector for the vocabulary mapping.

The probability of generating the next code token y is computed using the softmax function:

$$P(y) = \text{softmax}(\mathbf{z}) = \frac{\exp(\mathbf{z}_y)}{\sum_{j=1}^{|\mathcal{W}|} \exp(\mathbf{z}_j)},$$

where $\mathbf{z}_y$ is the score for the word y in the vocabulary, and $|\mathcal{W}|$ is the size of the vocabulary:

$$P(Y|T, D) = \prod_{i=1}^{m} P(y_i|T, D, y_1, y_2, \cdots, y_{i-1}).$$

The model generates a sequence of code tokens $y_1, y_2, \ldots, y_m$ by sampling from the probability distributions $P(y_1|T, D), P(y_2|T, D, y_1), \ldots, P(y_m|T, D, y_1, \ldots, y_{m-1})$, where each token is conditioned on the previous tokens in the sequence.

Prompt Compression. Prompt compression is a technique aimed at reducing the length of input prompts while preserving essential information. This method is particularly beneficial in scenarios where computational resources are constrained or where rapid response time is critical.

In a nutshell, the goal of prompt compression is to construct a function g, which, given the original prompt P, outputs the compressed prompt $g(P)$ such that the it contains less tokens but retains the critical information.

3 Empirical Study

In this section, we conduct an empirical study to explore the applicability of existing prompt compression methods to code generation tasks.

3.1 Experiment Setup

Datasets. We experiment on six diverse and widely adopted datasets which can simulate real-world code generation tasks. Our dataset selection follows two principles: (1) including dataset variants derived from the same base to evaluate robustness, and (2) covering different sources and complexity levels to ensure diversity.

For the first principle, we select HumanEval and its two variants from EvoEval (Subtle and Creative). While these datasets share similar base problems, they differ in how requirements are presented. This setup allows us to evaluate ShortenDoc's robustness in handling different expressions of the same functional requirement.

For the second principle, we select datasets from varied sources and at different complexity levels: MBPP contains human-written entry-level problems, CodeHarmony provides LLM-synthesized high-quality samples, and BigCodeBench presents practical and challenging programming tasks. The diversity ensures a comprehensive evaluation across different scenarios. In particular,

— *HumanEval* [6]: This dataset contains 164 well-designed hand-written programming problems for Python, each of which includes an average of 7.7 test cases per problem.
— *CodeHarmony* [12]: This dataset contains 16,153 high quality Python code samples synthesized by LLMs, which is extracted from existing open source datasets, such as the Evol dataset and OSS dataset. Each problem consists of three test cases. In our study, we choose CodeHarmony's test set containing 153 samples.
— *MBPP* [2]: This benchmark consists of around 1,000 crowd-sourced Python programming problems, designed to be solvable by entry level programmers, covering programming fundamentals, standard library functionality, and so on. Each problem consists of three automated test cases. In our study, we choose MBPP's test set containing 500 samples.
— *Subtle* [53]: This dataset contains 100 programming problems for Python based on HumanEval, which makes a subtle and minor change to the original problem such as inverting or replacing a requirement. Each problem consists of an average of 10.3 test cases per problem.
— *Creative* [53]: This dataset contains 100 programming problems for Python based on HumanEval, which generates a more creative problem compared to the original through the use of stories or uncommon narratives. Each problem consists of an average of 43.1 test cases per problem.
— *BigCodeBench* [58]: A set of 1,140 programming problems for Python which evaluates LLMs with practical and challenging programming tasks. Each problem consists of an average of 5.6 test cases. In our study, we choose BigCodeBench's hard set containing 148 samples.

Figure 1 shows the length distribution of DocString in these datasets. The different lengths distribution and diversity in these datasets ensure that our findings are applicable to a broad range of code generation scenarios, from simple script generation to complex software development tasks.

Study Models. We select a diverse set of LLMs that are representative of the current state-of-the-art in code generation, which include DeepSeekCoder-1.3b, DeepSeekCoder-6.7b [14], CodeQwen1.5 [3], CodeGeeX4 [57], and Llama3.1 [10].

— *DeepSeekCoder*: The 1.3b and 6.7b versions of DeepSeekCoder are trained on an extensive dataset, consisting of 87% code and 13% natural language in both English and Chinese. These models undergo pre-training on a project-level code corpus with a window size of 16K tokens, enhanced by a fill-in-the-blank task designed to bolster their project-level code completion and infilling capabilities.
— *CodeQwen1.5*: This model boasts 7.3 billion parameters and supports an impressive 92 programming languages, managing contexts up to 64K tokens with ease. It has demonstrated remarkable proficiency in code generation, long sequence modeling, code modification, and SQL capabilities.
— *CodeGeeX4*: With 9.4 billion parameters at its disposal, CodeGeeX4 is tailored for a myriad of AI software development scenarios. Its strengths lie in code completion, interpretation, web search, function calling, and repository-level question-and-answer interactions.

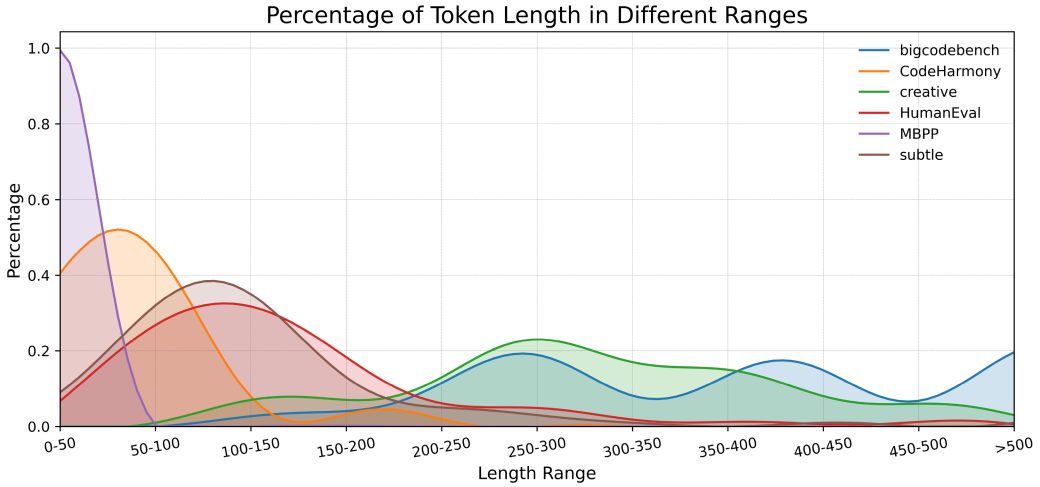

Fig. 1. The length distribution of DocString in these datasets. Notice for each dataset, the percentage represents the proportion of samples falling within a specific length range relative to the total number of samples. The length range indicates the number of tokens in the DocString after tokenization.

— *Llama3.1*: The 8.0 billion parameter version of Llama3.1 is an open source powerhouse, trained on a grand scale with the ability to handle context lengths up to 128K tokens.

Among these, DeepSeekCoder-1.3b, DeepSeekCoder-6.7b, CodeQwen1.5, and CodeGeeX4 are specialized for coding tasks, while Llama3.1, being a general-purpose model, has proven its versatility and potential for applications in diverse coding scenarios.

Existing Prompt Compression Methods. We explore two state-of-the-art prompt compression methods: Selective_Context [21] and LLMLingua2 [36].

Selective_Context employs a base language model (such as GPT-2) to compute the self-information of lexical units such as sentences, phrases, or tokens, which is then used to evaluate the informativeness. Specifically, it operates at the noun phrase level by first using NLTK for lexical analysis, then applying GPT tokenizer for subword tokenization, and finally merging tokens into noun phrases using spacy, which explains why some compressed tokens appear as subwords (e.g., "tu" for "tuples") in Figure 3.

LLMLingua2 is a task-agnostic prompt compression technique, i.e., it does not depend on the information entropy of the prompt. It utilizes data distillation to create an effective prompt compression dataset and trains a classification model as a way to select tokens that can be compressed. Unlike Selective_Context, LLMLingua2 operates at the complete word level, using a BERT classifier to identify redundant tokens.

Evaluation Metrics. For code generation Pass@1 and Compression Ratio are considered. Pass@1 indicates the percentage of tasks for which the model produces a code snippet that successfully passes all the associated test cases on the first try. This metric is a direct reflection of the model's accuracy and reliability in code generation scenarios. Compression Ratio quantifies the efficiency of the compression technique. It is defined as the percentage reduction in the length of the prompt after compression relative to the original length, i.e., $1 - \frac{\|D'\|}{\|D\|}$, where D is the original DocString and D' is the compressed one. A higher compression ratio indicates that more of the original prompt has been removed, which can lead to improvements in inference speed and reduced computational costs.

Implementation. All LLMs and their corresponding tokenizers are loaded from the official HuggingFace repository, ensuring that the up-to-date and optimized versions are used in our experiments. To ensure a fair comparison, we maintain the hyper-parameters of all models consistent throughout our study. For the backend service, we utilize the VLLM [17], which provides a unified interface for model inference. In addition, we employ Greedy Search as the inference strategy for all models. This method is chosen for its simplicity and effectiveness in generating sequential text, which is a common requirement in code generation tasks. Finally, the maximum output length for all models is set to 512 tokens.

3.2 Empirical Findings

In Table 1, we show the comparison of Pass@1 across different LLMs after applying Selective_Context and LLMLingua2 compression techniques. In particular, for each model on different datasets, it shows Pass@1 with the original prompts (0% compression) as well as the change in Pass@1 with increasing compression (10%, 20%, 30%, 40%).

(1) *Effectiveness.* As can be seen from the table, with a 10% compression rate, most models maintain comparable or even improved Pass@1 scores relative to their original performance. Taking the Pass@1 of CodeGeeX4 model on HumanEval dataset as an example, the original Pass@1 is 60.37%, the Pass@1 even can up to 62.20% at 10% compression rate using Selective_Context. Moreover, the Pass@1 can up to 62.80% at 10% compression rate using LLMLingua2.

However, this improvement is not universal across all datasets. Notably, on the BigCodeBench dataset, most models show a decrease in Pass@1 scores after compression. We attribute this phenomenon to the inherent complexity and diversity of DocStrings across different datasets.

These mixed results suggest that, while some DocStrings contain redundant information that can be safely compressed without compromising performance, the effectiveness of compression is highly dependent on the dataset characteristics. This observation highlights the importance of considering dataset-specific features when applying compression techniques.

(2) *Limitations.* As the compression rate increases further, the Pass@1 degradation trend becomes obvious. For example, the Pass@1 of the CodeGeeX4 model on the HumanEval dataset decreases to 49.39% at a compression rate of 40% using Selective_Context, and to 37.20% at a compression rate of 40% using LLMLingua2. This suggests that existing compression methods, while removing more information, may also remove semantic information that is critical for the model to generate correct code.

(3) *Code Generation after Removing DocString.* Interestingly, even in extreme cases, such as in the MBPP dataset, at 100% compression (i.e., DocString is completely removed), the models are still able to generate a certain percentage of correct code. This may indicate that in addition to DocString, other elements in the function signature, such as method name, carry important semantic information that is sufficient to guide the model in generating code that functional correct [9, 54]. This phenomenon is explored further in Section 6, as it relates to the deeper mechanisms of how the model utilizes different types of cue information to generate code.

Compressed Token Analysis. In addition to analyzing the compression ratio of prompt compression methods, we examine the specific tokens that are being removed. By reviewing the most frequent tokens that are compressed away, we can gain insights into the types of information that are considered redundant or less critical by the compression algorithms.

To this end, we extract the top-10 most frequently removed tokens with the highest frequency of occurrence for the two methods on the 10% compression ratio on the six datasets, respectively, and the results are shown in Table 2. We can find that the comparison between Selective_Context and LLMLingua2 reveals differences in the tokens each method targets for compression. This variation

Table 2. The Top-10 Most Frequently Removed Tokens in Each Dataset

Dataset	Rank-1	Rank-2	Rank-3	Rank-4	Rank-5	Rank-6	Rank-7	Rank-8	Rank-9	Rank-10
Selective_Context										
HE	of	2,	Output:	⇒	to	than	[1,	1,	[5,	3,
CH	of	to	with	on	Additionally,	returns	otherwise.	by	be	string.
MBPP	of	to	not.	list.	string.	array.	tuples.	in	from	number.
Subtle	of	2,	order.	descending	than	to	3,	[5,	order	by
Creative	3:	to	Output:	on	than	2:	2,	ascending	4,	(1,
BCB	be	on	of	Notes:	not	ValueError:	which	[1,	matplotlib.	2,
LLMLingua2										
HE	the	that	a	and	are	The	an	You	of	is
CH	the	a	The	are	that	and	an	This	where	which
MBPP	the	a	that	to	an	are	and	is	of	which
Subtle	the	a	that	and	are	an	The	of	is	any
Creative	The	that	the	an	a	are	and	there	which	where
BCB	The	a	the	that	an	then	A	and	are	there

could be due to the distinct algorithms and criteria each method uses to evaluate the importance of tokens.

Specifically, Selective_Context, which is based on information entropy, focuses more on the semantic coherence of the text. It tends to prioritize the retention of tokens that contribute to the logical flow and understanding of the code, thus it might be less likely to compress away tokens that are crucial for the structure or functionality of the code, even if they are not the most frequent.

On the other hand, LLMLingua2, which utilizes a pre-trained classifier, is more attuned to the significance of information. It is particularly sensitive to articles and prepositions such as "the" and "a," which are common in English but may not carry substantial meaning in the context of code. This method might prioritize compressing these tokens as they are less likely to affect the overall functionality or readability of the code.

The divergence in the tokens targeted by each method underscores the importance of the underlying algorithm's design in determining what constitutes "redundant" information. It also highlights the need for a nuanced approach to prompt compression, where the method must balance the reduction of redundancy with the preservation of critical information necessary for code comprehension and execution.

4 Approach

Building upon insights from the empirical research, we propose a novel compression technique ShortenDoc, which is shown in Figure 2. Our approach consists of four main stages: Step 1: Pre-Processing, Step 2: Token Importance Sort, Step 3: Search Space Construction, and Step 4: Constraint Definition.

Given a prompt consisting of a signature T and a DocString D, our goal is to compress the DocString into D' while preserving the overall semantics and effectiveness in guiding LLMs for code generation.

We formalize this as an optimization problem aiming to find the optimal compressed DocString D' that satisfies semantic similarity constraints C (which we discuss in Section 4.4). The optimization task is defined as:

$$\begin{aligned} \underset{D'}{\text{minimize}} \quad & \|D'\| \\ \text{subject to} \quad & C, \text{and } D' \sqsubseteq D, \end{aligned}$$

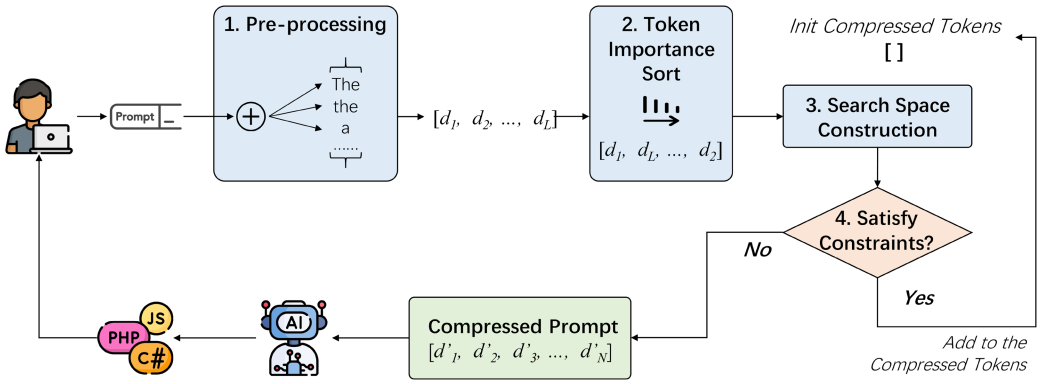

Fig. 2. Framework of ShortenDoc.

where $\|D'\|$ denotes the length (number of tokens) of the compressed DocString D', and $D' \sqsubseteq D$ asserts that D' is a *subsequence* of the original DocString D.

Specifically, in our approach, we opt for Byte Pair Encoding tokenizer's subword level, which aligns with the native tokenization of most modern LLMs while providing fine-grained compression control, and eliminates the need for additional token conversion steps, reducing computational overhead. Furthermore, subword tokenization offers finer-grained control over compression. For instance, when compressing technical terms or compound words in DocStrings, subword-level operations can preserve critical parts while removing less essential components.

4.1 Pre-Processing

Inspired by the previous study [48], we posit that the removal of line breaks and tabs from DocString does not significantly affect Pass@1 of the code generation model. (An analysis will be discussed in Section 6.)

Moreover, upon examining the tokens that frequently undergo compression, as indicated in Table 2, we observe that the tokens compressed by LLMLingua2 exhibit regular patterns (such as "the," "a"), and the model's Pass@1 score remains largely intact when compression is applied at a rate of 10%. This insight suggests that the compression of these tokens does not adversely affect model's Pass@1 score, leading us to categorize them as stop-words.

To minimize semantic distortion in the compressed DocString, we calculate the semantic similarity between the original and the compressed versions for each stop word we attempt to compress. This ensures that the meaning of the DocString remains closely aligned with the original, even after compression. Specifically, we choose the CodeGPT-py-adapted [24] and adopt the methodology proposed by OpenAI [31] for extracting semantic vectors corresponding to the text, followed by the computation of semantic similarity using cosine similarity. Compression is performed only for those tokens where the similarity metric exceeds a threshold of 0.999, ensuring the preservation of DocString's semantic integrity.

4.2 Token Importance Sort

Following the pre-processing phase, we obtain the token sequence $D = [d_1, \ldots, d_L]$. To determine the importance of each token in this sequence, we draw upon principles from information theory. Specifically, the significance of each token d_t is quantified by its contribution to the conditional entropy of the sequence, which is quantified by its negative log-probability given its

preceding context:

$$\text{Information}(d_t) = -\log P(d_t \mid d_1, d_2, \cdots, d_{t-1}).$$

This measure reflects how informative the token d_t is, given the tokens that come before it. A higher value indicates that the token carries more information and is thus more important to the sequence's overall meaning.

To calculate the distribution P, we also adopt CodeGPT-py-adapted [24] as the foundational language model for assessing token significance and formulating subsequent constraints. This choice is justified by the model's relatively modest parameter count and minimal GPU memory requirements, which facilitate efficient computation. Furthermore, as demonstrated in LLMLingua [15], smaller language models such as GPT2-small can maintain comprehension of compressed prompts.

The perplexity of the entire DocString D serves as a measure of its overall information content, calculated as the average conditional entropy across all tokens:

$$\text{Perplexity}(D) = -\frac{1}{L}\sum_{i=1}^{L}\log P(d_i|d_1, d_2, \cdots, d_{i-1}).$$

This value serves as a measure of the sequence's overall unpredictability or information content.

To elucidate the individual informational contribution of each token, we compute the perplexity of D^{-i}, obtained by excluding a specific token d_i from D, viz., $D^{-i} = (d_1, d_2, \ldots, d_{i-1}, d_{i+1}, \ldots, d_L)$ The importance of each token d_i is then calculated as the difference between the perplexity of D and that of D^{-i}, formulated as:

$$\text{Importance}(d_i) = \text{Perplexity}(D) - \text{Perplexity}(D^{-i}).$$

We then sort the tokens in ascending order based on their importance scores, i.e., $D' = [d_{i_1}, \cdots, d_{i_L}]$ where $i_1, i_2, \ldots, i_L$ represent the index of each token in D.

This approach ensures that tokens whose removal leads to a significant increase in the sequence's conditional entropy are identified as highly important. These tokens are critical for maintaining the semantic integrity of the DocString and are therefore prioritized for retention during the compression process. Conversely, tokens whose removal results in minimal changes to the conditional entropy are considered less important. These tokens are prime candidates for compression, as their absence is unlikely to adversely affect the model's understanding of the DocString's content.

4.3 Search Space Construction

After ranking the tokens by their importance scores, we proceed to construct the search space $\mathcal{S}$ which comprises candidate tokens (sequences) that may be removed. To efficiently explore the search space and avoid suboptimal solutions due to local optima, we employ a Top-N strategy, i.e., to select the top N least important tokens $[d_{i_1}, d_{i_2}, \ldots, d_{i_N}]$. Our empirical observations in Table 1 indicate that, in certain scenarios, a higher compression rate may yield enhanced Pass@1 score. Taking the DS-6.7B model as an example, on the CodeHarmony dataset, the Pass@1 of LLMLingua2 under 20% compression outperforms that under 10% compression. We hypothesize that the simultaneous compression of consecutive tokens could provide unexpected benefits over compressing individual tokens. As a result, for each $1 \le k \le N$ we consider k-gram, i.e.,

$$\mathcal{G}_k = \{[d_{i_j}, d_{i_{j+1}}, \ldots, d_{i_{j+k-1}}] \mid 1 \le j \le N - k + 1\}.$$

Each $\mathcal{G}_k$ represents the set of all possible consecutive k-grams within the Top-N tokens.

The complete search space $\mathcal{S}$ is the union of all $\mathcal{G}_k$:

$$\mathcal{S} = \bigcup_{k=1}^{N} \mathcal{G}_k.$$

This approach considers both individual low-importance tokens and sequences of such tokens, thereby exploring a richer set of compression candidates.

In terms of computational complexity, the number of combinations for each k-gram is $N - k + 1$, so the size of $\mathcal{S}$ is bounded by $\frac{N(N+1)}{2}$.

For example, let us assume that $N = 5$ and $\mathcal{I} = [2, 3, 10, 13, 15]$, then the search space can be defined as:

— 1-grams: $\mathcal{G}_1 = \{[2], [15], [3], [10], [13]\}$,
— 2-grams: $\mathcal{G}_2 = \{[2, 15], [15, 3], [3, 10], [10, 13]\}$,
— 3-grams: $\mathcal{G}_3 = \{[2, 15, 3], [15, 3, 10], [3, 10, 13]\}$,
— 4-grams: $\mathcal{G}_4 = \{[2, 15, 3, 10], [15, 3, 10, 13]\}$,
— 5-grams: $\mathcal{G}_5 = \{[2, 15, 3, 10, 13]\}$.

By including N-gram combinations, we capture the potential benefits of compressing sequences of tokens that may collectively have a low impact on the semantic content but offer greater compression efficiency when removed together. This comprehensive search space allows us to explore various compression strategies and select the one that best satisfies our optimization objectives and constraints.

4.4 Constraint Definition

In the context of code generation, the probability of generating a sequence of code Y by given a signature T and a DocString D is defined as:

$$P(Y|T, D) = \prod_{i=1}^{m} P(y_i|T, D, y_1, y_2, \ldots, y_{i-1}).$$

Here, T is considered constant for different compressions, implying that the variability in $P(Y|T, D)$ is attributed to changes in D. We denote the compressed version of D as D', and the corresponding probability as:

$$P(Y|T, D') = \prod_{i=1}^{m} P(y_i|T, D', y_1, y_2, \ldots, y_{i-1}).$$

Our goal is to minimize the difference between $P(Y|T, D)$ and $P(Y|T, D')$, ensuring that the compression does not significantly alter the model's output:

$$P(Y|T, D) \approx P(Y|T, D').$$

However, directly computing the divergence between these two distributions over all possible code sequences $\mathcal{Y}$ is computationally infeasible. To address this, we approximate the requirement by focusing on the model's output distribution for the next token. Given the sequential nature of code generation, where each token depends on the previous tokens, we simplify the comparison to the probability of generating the first token y_1:

$$P(y_1|T, D) \approx P(y_1|T, D').$$

The probability of the first token is given by the softmax function of the model's output:

$$P(y_1|T, D) := \text{softmax}(z(T, D)).$$

Thus, the constraint can be expressed as the proximity between the softmax inputs for the original and compressed sequences:

$$\text{softmax}(z(T, D)) \approx \text{softmax}(z(T, D')).$$

To ensure that the model's behavior remains consistent after compression, we impose a constraint on the similarity between the logits $z(T, D)$ and $z(T, D')$. We use the cosine similarity metric to quantify the closeness between these two vectors:

$$\text{CosineSimilarity}(z(T, D), z(T, D')) = \frac{z(T, D) \cdot z(T, D')}{\|z(T, D)\| \cdot \|z(T, D')\|}.$$

Our constraint is then formalized as:

$$\text{CosineSimilarity}(z(T, D), z(T, D')) \geq \tau.$$

Here, τ is the hyper-parameter which is set to be 0.999 in our study.

Integrating this constraint into our overall optimization framework, we aim to find the compressed DocString D' that minimizes its length while satisfying the semantic similarity constraint:

$$\begin{aligned} \underset{D'}{\text{minimize}} \quad & \|D'\| \\ \text{subject to} \quad & D' \sqsubseteq D, \\ & \text{CosineSimilarity}(z(T, D), z(T, D')) \geq \tau. \end{aligned}$$

5 Evaluation

We evaluate the effectiveness of our proposed approach from three perspectives.

5.1 RQ1: What Is the Effectiveness of Our Proposed ShortenDoc Compared to Existing Prompt Compression Methods? (Pass@1 Comparison)

We aim to evaluate the effectiveness of ShortenDoc in reducing DocString length without compromising the Pass@1 score of code generation models. To ensure the robustness of our findings, we also include a closed-source state-of-the-art model *GPT-4o* in our analysis. Additionally, we introduce the Random method as a baseline. The focus of our analysis is on the Pass@1 score and the compression ratio across a spectrum of datasets and models, as presented in Table 3.

The compression ratios for different datasets are carefully determined through preliminary experiments, considering the following factors:

- *Dataset Characteristics*: Different datasets exhibit varying DocString patterns and information density. For instance, MBPP's DocStrings contain more redundant information, allowing a higher compression ratio (38%), while CodeHarmony's more concise DocStrings necessitate a lower ratio (25%).
- *Fair Comparison*: To ensure a fair comparison, we apply the same compression ratio to all methods within each dataset. While different methods might achieve their optimal performance at different ratios, using consistent ratios within datasets allows a direct comparison of compression effectiveness.

Our results demonstrate that ShortenDoc consistently outperforms alternative methods across all datasets and models, particularly at higher compression rates. For instance, on the HumanEval dataset, ShortenDoc achieves the highest Pass@1 score across all models, effectively maintaining or even improving code generation quality despite a 30% reduction in DocString length. This indicates that ShortenDoc can sustain, and in some cases enhance, model Pass@1 score while significantly compressing input prompts.

Table 3. Pass@1 Comparison of Different Methods across Various Datasets and Models

Dataset	Method	Ratio	LLMs					
			DS-1.3B	DS-6.7B	CQ-7.3B	CG-9.4B	LA-8.0B	GPT-4o
HumanEval	None	0	63.42	71.95	77.44	60.37	57.32	85.37
	Random	30	32.32	38.41	60.37	36.59	35.37	50.61
	Selective_Context	30	44.51	61.59	70.12	56.10	53.05	75.00
	LLMLingua2	30	42.68	54.27	65.24	52.44	45.12	67.68
	ShortenDoc	30	**57.93**	**72.56**	**78.66**	**64.34**	**56.10**	**83.54**
CodeHarmony	None	0	59.48	64.36	60.78	64.71	59.48	62.09
	Random	25	49.63	60.78	51.63	52.29	56.21	54.25
	Selective_Context	25	54.90	67.32	56.86	59.48	59.48	61.44
	LLMLingua2	25	54.25	**67.97**	61.40	59.48	**62.09**	61.44
	ShortenDoc	25	**55.56**	66.01	**63.40**	**60.78**	61.44	**62.09**
MBPP	None	0	36.40	46.20	59.00	46.60	41.40	55.00
	Random	38	30.20	39.40	48.20	38.60	32.00	46.20
	Selective_Context	38	34.60	**44.80**	**55.40**	42.00	37.60	50.00
	LLMLingua2	38	31.00	42.00	52.40	42.20	37.20	50.00
	ShortenDoc	38	**36.60**	43.80	**55.40**	**46.20**	**41.40**	**51.40**
Subtle	None	0	53.00	61.00	63.00	64.00	56.00	81.00
	Random	30	28.00	35.00	40.00	35.00	34.00	42.00
	Selective_Context	30	45.00	55.00	52.00	52.00	50.00	64.00
	LLMLingua2	30	44.00	45.00	55.00	51.00	47.00	60.00
	ShortenDoc	30	**54.00**	**56.00**	**62.00**	**66.00**	**56.00**	**79.00**
Creative	None	0	23.00	34.00	38.00	36.00	34.00	54.00
	Random	25	9.00	16.00	18.00	21.00	21.00	29.00
	Selective_Context	25	19.00	28.00	29.00	32.00	28.00	49.00
	LLMLingua2	25	**23.00**	33.00	29.00	31.00	**35.00**	51.00
	ShortenDoc	25	**23.00**	**36.00**	**37.00**	**35.00**	**35.00**	**58.00**
BigCodeBench	None	0	6.10	12.20	13.50	14.20	10.80	27.70
	Random	34	0.00	5.40	4.70	5.40	1.40	12.20
	Selective_Context	34	**2.00**	**6.10**	6.10	8.10	2.70	15.50
	LLMLingua2	34	**2.00**	5.40	6.10	8.80	4.10	15.50
	ShortenDoc	34	**2.00**	**6.10**	**10.10**	**9.50**	**4.70**	**20.30**

Bold means the best performance in the comparison experiment.

Moreover, in several cases, ShortenDoc surpasses the uncompressed baseline. For instance, on the HumanEval dataset, DeepSeekCoder-6.7B, CodeQwen, and CodeGeeX4 can even achieve a higher Pass@1 score when compressed by ShortenDoc. This surprising result suggests that ShortenDoc not only preserves essential information but may also eliminate redundant or distracting content from the DocStrings, thereby improving model's focus on relevant information during code generation.

Consistently, across other datasets such as CodeHarmony, MBPP, Subtle, Creative, and BigCodeBench, ShortenDoc either matches or exceeds the competing methods, especially under aggressive compression scenarios. The uniform superiority of ShortenDoc across diverse datasets and models underscores its versatility and robustness. These results suggest that ShortenDoc effectively balances compression and information retention, making it a potent tool for optimizing prompt inputs in code generation tasks.

Table 4. FLOPs Comparison across Various Datasets and Models

Dataset	DS-1.3B		DS-6.7B		CQ-7.3B		CG-9.4B		LA-8.0B	
	Compress	Raw	Compress	Raw	Compress	Raw	Compress	Raw	Compress	Raw
HumanEval	0.28(+0.02)	0.34	1.43(+0.02)	1.74	1.43(+0.02)	1.79	1.70(+0.02)	2.02	1.46(+0.02)	1.73
CodeHarmony	0.12(+0.01)	0.15	0.62(+0.01)	0.75	0.63(+0.01)	0.78	0.75(+0.01)	0.88	0.65(+0.01)	0.75
MBPP	0.03(+0.01)	0.04	0.13(+0.01)	0.22	0.14(+0.01)	0.22	0.18(+0.01)	0.28	0.15(+0.01)	0.24
Subtle	0.25(+0.02)	0.30	1.31(+0.02)	1.55	1.29(+0.02)	1.59	1.53(+0.02)	1.77	1.31(+0.02)	1.52
Creative	0.65(+0.02)	0.75	3.37(+0.02)	3.89	3.35(+0.02)	3.90	3.97(+0.02)	4.41	3.39(+0.02)	3.75
BigCodeBench	0.70(+0.05)	0.85	3.59(+0.05)	4.39	3.53(+0.05)	4.44	4.00(+0.05)	4.67	3.41(+0.05)	3.98

The brackets indicate the FLOPs needed for the ShortenDoc calculation.

Furthermore, the fact that ShortenDoc maintains a high Pass@1 score across models of varying sizes and architectures indicates its generalizability. It effectively aids both open source and closed-source models, including state-of-the-art systems like GPT-4o. This broad applicability enhances the practical value of ShortenDoc in real-world code generation applications.

Efficiency. To comprehensively evaluate the efficiency of ShortenDoc, we analyze its performance from three perspectives:

(1) *GPU Memory Usage*: GPU memory consumption is a critical resource constraint in deploying LLMs for code generation tasks. To evaluate the memory efficiency of our approach, we compare the GPU memory usage between using compressed and uncompressed DocStrings.

It is worth noting that, while our method employs CodeGPT-py-adapted as the base model for compression, its memory footprint is negligible compared to the deployment of LLMs. Specifically, CodeGPT-py-adapted has only 124M parameters and requires merely 0.23 GB of GPU memory under the torch.bfloat16 precision. This overhead is insignificant when compared to the substantial memory requirements of modern LLMs used for code generation.

(2) *FLOPs Analysis*: FLOPs refers to the number of floating-point operations, an indicator used to measure the complexity of a model, reflecting the total number of FLOPs required during its execution. FLOPs is closely related to the input length and structure of the model. The purpose of DocString compression is to reduce computational costs and enhance model efficiency. Therefore, we utilize the FLOPs metric to evaluate the computational load of different models before and after compression across various datasets.

The results in Table 4 indicate that the computational load of the models generally decreases after compression. It can be observed that the FLOPs values for the compressed models (Compress) are lower than those of the original models (Raw) across all datasets. For instance, on the HumanEval dataset, the FLOPs decreased from 0.34 to 0.28 after compression, which is a reduction of approximately 17.6%. On the CodeHarmony dataset, the FLOPs decreased from 0.15 to 0.12 after compression, which is a reduction of about 20%. This reduction in computational load due to compression means that under the same hardware conditions, the model can process data more quickly, or process more data in the same amount of time.

(3) *Inference Time*: End-to-end inference time is a crucial metric for real-world applications, directly impacting user experience and system throughput. To assess the practical benefits of our compression approach, we conduct comprehensive timing measurements under controlled experimental conditions using NVIDIA RTX 3090 GPU. All models were configured to use the torch.bfloat16 precision with both input and output lengths set to 512 tokens.

The results show that the code generation time varies significantly across different model scales. The DeepSeekCoder model with 1.3B parameters requires approximately 9.21 seconds per generation, while larger models around 7B parameters take about 17.77 seconds. For comparison,

when using the GPT-4 API, the generation time typically ranges from 5 to 10 seconds, though it may fluctuate depending on the server load.

The time overhead introduced by ShortenDoc's compression process is relatively modest, averaging around 2 seconds per DocString. This additional pre-processing time proves to be a worthwhile investment. In practical applications, the total processing pipeline (compression plus inference) remains highly efficient.

Summary of RQ1

ShortenDoc consistently outperforms existing compression methods across multiple datasets and models, effectively reducing DocString length without compromising, and sometimes even enhancing code generation Pass@1 score.

5.2 RQ2: What Is the Effectiveness of Our Proposed ShortenDoc in Other Program Languages? (Generalization Capability)

Considering that the datasets chosen in RQ1 are all in Python, to explore the generalizability of ShortenDoc, we select four other programming languages, i.e., C++, Go, Java, JavaScript in the HumanEval-X [57] dataset for our experiments.

Our choice of HumanEval-X is primarily driven by the status quo. At present, most publicly available code generation datasets for non-Python languages are variants of HumanEval, as it has become a *de facto* benchmark in the field. While we acknowledge that using variations of a single dataset might introduce potential biases, HumanEval-X represents one of the few comprehensive, well-validated multi-lingual code generation benchmarks available to the research community.

Since these four datasets share DocString with HumanEval, we directly migrate the compressed DocString obtained for HumanEval in RQ1 to these datasets. The detailed experimental results are shown in Table 5.

The results reveal that ShortenDoc maintains its superior performance across all four programming languages when compared to baseline methods (e.g., Random compression, Selective_Context, and LLMLingua2). Despite the structural and syntactical differences among these languages, ShortenDoc consistently achieves higher performance metrics, demonstrating its generalization ability to retain the critical information in DocStrings.

Notably, in the C++ and Java datasets, ShortenDoc significantly outperformed other methods by maintaining closer alignment with the uncompressed baseline, indicating that its approach to token importance and compression strategy is not tightly coupled to any specific programming language. The JavaScript dataset, which tends to have more varied DocString usage, still saw considerable gains with ShortenDoc, reinforcing its versatility. The Go dataset showed a similar trend, where ShortenDoc consistently outperformed the baselines and adapted well to the more concise and functionally focused nature of Go's syntax.

At the same time, we observe a slight degradation in Pass@1 scores when directly migrating the compressed DocStrings obtained in RQ1 (Python) to other programming languages such as C++, Go, Java, and JavaScript. While ShortenDoc consistently outperformed baseline methods, its performance did not always fully match the results achieved with the original uncompressed DocStrings in these new languages.

This performance degradation can be attributed to the fact that the base language model used in our approach, CodeGPT, is specifically trained and optimized for Python. The model's understanding of the syntax, structure, and semantics of Python-based code influences how DocStrings are compressed. When these compressed DocStrings are applied to other programming languages,

Table 5. Generalization Capability of ShortenDoc in Different Programming Languages

Dataset	Method	Ratio	LLMs					
			DS-1.3B	DS-6.7B	CQ-7.3B	CG-9.4B	LA-8.0B	Avg.
CPP	-	0	47.56	63.41	65.85	59.76	43.90	56.10
	Random	30	26.22	37.20	41.46	34.15	29.27	33.66
	Selective_Context	30	39.63	52.44	57.32	43.29	37.80	46.10
	LLMLingua2	30	33.54	50.00	55.49	43.90	31.71	42.93
	ShortenDoc	30	**42.07**	**58.54**	**60.37**	**58.54**	**39.02**	**51.71**
Go	-	0	46.95	61.59	58.54	52.44	37.80	51.46
	Random	30	21.95	32.32	35.37	27.44	22.56	27.93
	Selective_Context	30	**44.51**	53.05	59.76	40.24	27.44	45.00
	LLMLingua2	30	34.76	42.07	55.49	36.59	26.22	39.02
	ShortenDoc	30	39.63	**56.10**	**64.63**	**46.95**	**31.10**	**47.68**
Java	-	0	57.93	59.15	77.44	60.37	57.93	62.56
	Random	30	37.20	44.51	54.27	37.80	31.10	40.98
	Selective_Context	30	**55.49**	58.54	71.95	57.32	45.73	57.80
	LLMLingua2	30	46.95	51.83	67.68	54.27	34.15	50.98
	ShortenDoc	30	53.66	**67.68**	**75.00**	**60.37**	**48.17**	**60.98**
JavaScript	-	0	57.93	64.63	71.34	59.15	40.85	58.78
	Random	30	27.44	43.90	49.39	35.37	26.22	36.46
	Selective_Context	30	46.34	55.49	57.32	43.29	**46.34**	49.76
	LLMLingua2	30	45.12	48.17	**66.46**	47.56	31.71	47.80
	ShortenDoc	30	**56.10**	**67.07**	65.24	**49.39**	41.46	**55.85**

Bold means the best performance in the comparison experiment.

certain language-specific nuances may not be as effectively preserved, leading to a small drop in code generation performance.

Despite this limitation, ShortenDoc still demonstrates strong generalization capability, retaining the critical information needed for code generation tasks across multiple programming languages. This suggests that while language-specific models might yield the best results when paired with ShortenDoc, the method remains highly effective even when applied across different languages with minimal adjustments.

Summary of RQ2

While ShortenDoc generalizes well across programming languages, a slight performance degradation is observed when migrating compressed DocStrings from Python to other languages. This can be attributed to the base model, CodeGPT, being trained specifically for Python, which may not fully capture the nuances of other languages. Nonetheless, the generalization capability of ShortenDoc was thoroughly demonstrated by its performance across C++, Go, Java, and JavaScript in the HumanEval-X dataset. By directly migrating compressed DocStrings from Python datasets, ShortenDoc consistently outperformed baseline methods, proving its adaptability and robustness across multiple programming languages.

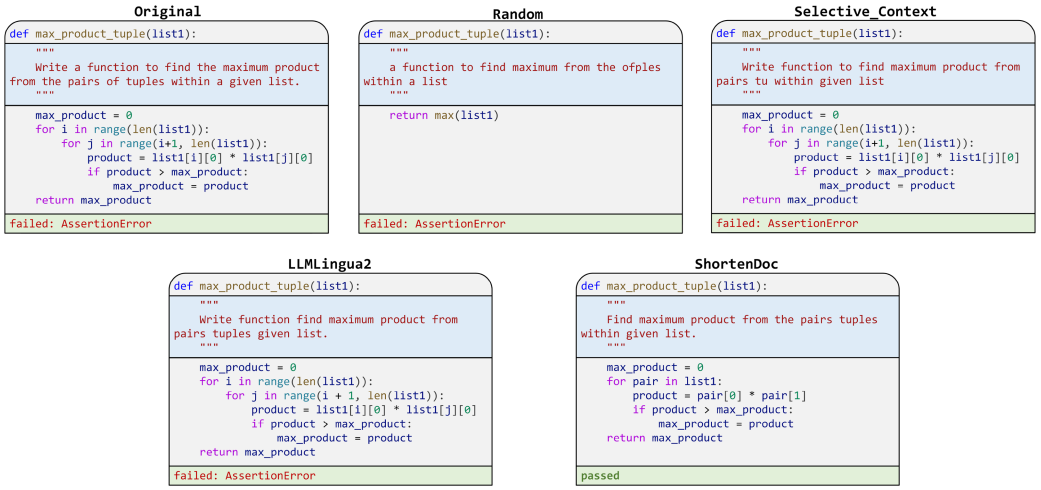

Fig. 3. Example of different DocString compression methods.

5.3 RQ3: What Is the Impact of Compressed Prompts on Developers? (Human Study)

RQ1 and RQ2 rely on automated evaluation metrics to compare the performance of different methods. In RQ3, we aim to gain more insights by conducting a human study to assess the impact of DocString compression.

To complement our analysis in RQ1 and RQ2, we first present a single example to provide deeper insights into how different compression methods affect code generation. While a single example cannot represent all scenarios, it helps illustrate the typical patterns we have observed across our experiments. Figure 3 showcases a DocString from the Llama3.1 model within one representative example from the MBPP dataset, showing a DocString processed by different compression methods. We display the original DocString alongside its compressed versions, processed by four distinct methods. The original DocString led to semantically incorrect code generation by the Llama3.1 model. The Random compression method resulted in irrelevant code. Both Selective_Context and LLMLingua2 maintained the model's output, yet failed to rectify the semantic inaccuracy. In stark contrast, our tool's compression successfully steered the model towards semantically correct code generation.

To further evaluate the quality of the compressed DocStrings, we conducted a human study using three groups of DocStrings compressed by different methods (Selective_Context, LLMLingua2, and ShortenDoc).

By involving human evaluators, we sought to provide a more comprehensive assessment of the compressed DocStrings and to better understand their practical implications. In our human study, we used two key evaluation criteria: Informativeness and Comprehensibility.

— *Informativeness*. This criterion measures whether the compressed DocString retains the key information conveyed in the original DocString. A higher score indicates that the essential content is preserved after compression.
— *Comprehensibility*. This criterion evaluates how easily a human can understand the compressed DocString. It measures whether the DocString, despite being shortened, remains understandable.

Reading the following DocStrings and answer the questions:

Prompt:

```
def max_product_tuple(list1):
    """
    Write a function to find the maximum product from the pairs of tuples within a
given list.
    """
```

Candidate 1:

```
    """
    Write function to find maximum product from pairs tu within given list
    """
```

Candidate 2:

```
    """
    Write function find maximum product from pairs tuples given list.
    """
```

Candidate 3:

```
    """
    Find maximum product from the pairs tuples within given list.
    """
```

Evaluate (*Score 0 to 4, 4 is the best*)	**Candidate 1**	**Candidate 2**	**Candidate 3**
Informativeness			
Comprehensibility			

Fig. 4. A sample questionnaire used in human study. The "Prompt" section shows the original uncompressed DocString, while the "Candidates" section presents three compressed versions of the same DocString generated by different methods (Selective_Context, LLMLingua2, and ShortenDoc). Evaluators were asked to rate each candidate on Informativeness (preservation of key information) and Comprehensibility (ease of understanding) using a 0–4 scale, without knowing which method produced each compression.

To conduct the human study, we recruited three evaluators with a strong background in Python programming (all are PhD candidates). We randomly selected 40 samples from each of the six datasets used in the previous experiments, resulting in a total of 240 samples for evaluation. For each sample, the evaluators assessed the original DocString along with three compressed versions generated by different methods.

To facilitate analysis and comparison across different data sources, we organized the samples into six groups according to their source datasets. Each group was evaluated by all three evaluators. The evaluators assessed each DocString anonymously based on two criteria: Informativeness and Comprehensibility. The evaluation scale ranged from 0 to 4 points, with higher scores indicating better quality. The final score for each criterion was calculated as the average of scores from all three evaluators.

The questionnaire used for the study is illustrated in Figure 4. To maintain the quality of the human evaluation, the compressed DocStrings were presented in random order, ensuring that the evaluators were unaware of which method generated the DocString. Additionally, the evaluators were allowed to use Internet to look up unfamiliar concepts. To prevent fatigue and maintain concentration, we asked each evaluator to complete the assessment of one dataset (40 samples) over 2 days, with no more than 20 samples evaluated in any half-day session.

The results presented in Table 6 from the human study clearly demonstrate that the proposed ShortenDoc method outperforms the other two methods, Selective_Context and LLMLingua2, in both Informativeness and Comprehensibility. This superiority indicates that ShortenDoc is more

Table 6. The Average Score and Kappa Score (in Parentheses) of Human Study

Aspect	Method	Datasets					
		HumanEval	CodeHarmony	MBPP	Subtle	Creative	BigCodeBench
Informativeness	Selective_Context	2.83 (0.4)	3.37 (0.5)	**3.22** (0.2)	2.85 (0.3)	3.55 (0.6)	3.60 (0.5)
	LLMLingua2	2.85 (0.5)	3.32 (0.4)	2.83 (0.5)	2.80 (0.5)	3.55 (0.5)	3.35 (0.5)
	ShortenDoc	**3.83** (0.7)	**3.75** (0.7)	3.13 (0.5)	**3.77** (0.7)	**3.97** (0.9)	**3.96** (0.9)
Comprehensibility	Selective_Context	1.79 (0.2)	2.44 (0.1)	2.86 (0.1)	1.77 (0.2)	1.87 (0.1)	1.90 (0.0)
	LLMLingua2	2.30 (0.4)	2.92 (0.3)	2.74 (0.3)	2.24 (0.4)	2.85 (0.2)	2.68 (0.2)
	ShortenDoc	**3.48** (0.3)	**3.50** (0.3)	**3.61** (0.2)	**3.33** (0.3)	**3.67** (0.2)	**3.66** (0.1)

Bold means the best performance in the comparison experiment.

effective in retaining key information and maintaining the understandability of DocStrings after compression.

Furthermore, to appraise the consistency among the three evaluators, we computed Fleiss' Kappa statistic [13, 40]. Fleiss' Kappa is a generalization of Cohen's Kappa that is used to measure the reliability of categorical data when there are multiple raters. It provides a measure of the degree to which the evaluators agree beyond what would be expected by chance. A Kappa value of 1 indicates perfect agreement, while a value of 0 suggests no agreement beyond chance. Negative values indicate agreement less than what would be expected by chance. In our study, a Kappa coefficient was calculated to ensure that the evaluation scores were consistent and reliable across the evaluators.

In terms of Informativeness, the ShortenDoc method consistently scores higher than the other two methods across most datasets. This suggests that ShortenDoc is adept at preserving the essential information within the DocStrings even after compression. In addition, the high Fleiss' Kappa scores, ranging from 0.5 to 0.9, indicate a strong consensus among the evaluators regarding the Informativeness of the compressed DocStrings. This high agreement likely stems from the objective nature of this metric, where evaluators can clearly discern whether the compressed DocString retains the necessary information from the original.

In terms of Comprehensibility, ShortenDoc also shows superiority than the other two methods in Comprehensibility, indicating that the compressed DocStrings remain understandable to human readers. While the lower Fleiss' Kappa scores for Comprehensibility, ranging from 0.1 to 0.3, suggest that there is less consistency among the evaluators in this metric compared to Informativeness.

The distinct scores in evaluator agreement between Informativeness and Comprehensibility can be attributed to their fundamental differences: (1) Subjectivity. Comprehensibility is inherently more subjective than Informativeness, as it depends on the evaluator's personal ability to understand the compressed information. (2) Complexity of Docstrings. Some DocStrings, despite being compressed, may still contain complex information or jargon that is more challenging for some evaluators to grasp. (3) Diversity in Evaluator Background. The professional background and experience of the evaluators can influence their assessment of DocString comprehensibility. For instance, evaluators with specific domain knowledge might find certain terms or concepts more comprehensible.

Summary of RQ3

The human study confirms that ShortenDoc excels in both Informativeness and Comprehensibility compared to Selective_Context and LLMLingua2. The strong performance of ShortenDoc across both dimensions demonstrates that it not only compresses DocStrings effectively but also preserves critical information and maintains clarity, making it a practical solution for real-world applications.

6 Discussion

In this section, we delve into the insights gained from the DocString compression techniques and explore the patterns that emerged from our experiments.

6.1 The Impact of Method Name Quality

In Section 3, we observed an intriguing phenomenon: Models were capable of generating some correct code without relying on DocStrings, indicating that the method signature itself, particularly the method name, carries substantial information. This observation led us to hypothesize that the quality of method names plays a pivotal role in the success of DocString compression.

To empirically validate this hypothesis, we designed a controlled experiment. We systematically replaced high-quality method names with a generic placeholder, such as "foo" [54], across various datasets including HumanEval, CodeHarmony, MBPP, Subtle, and Creative.

Conversely, we enhanced the BigCodeBench dataset by upgrading the low-quality method names to more descriptive and informative alternatives. We observed that all method names in the dataset were generic, lacking any meaningful information, such as "func_task." To address this, we manually replaced each sample with high-quality method names that we crafted to provide better context and clarity.

The outcomes of this experiment are encapsulated in Table 7. The data clearly demonstrate that method names wield a significant influence on the Pass@1 score of DocString compression. Across the board, models exhibited a notable decline in Pass@1 score when high-quality method names were replaced with the generic "foo." This finding suggests that a portion of the information compressed by ShortenDoc is redundant with the information already present in the method name.

Conversely, when low-quality method names were enhanced, we observed a consistent uptick in model Pass@1 score. This improvement underscores the notion that enriching the quality of method names not only aids in the comprehension of the code but also bolsters the robustness of ShortenDoc's compression capabilities.

In light of these findings, we advocate for a conscientious approach to method name quality during the DocString compression process. By elevating the quality of method names, we can potentially enhance the clarity of the code and the effectiveness of the compression algorithm. This strategy may yield unexpected benefits, such as more efficient code generation and improved maintainability of the codebase.

6.2 The Impact of DocString Style

The style of DocStrings is a critical yet often overlooked aspect of code documentation. In our investigation, we explored the impact of DocString style on the performance of our compression tool and the models' ability to generate accurate and understandable code. Our analysis led to several key findings and considerations.

Experiments with Newline and Tab Characters. In Section 4, during the pre-processing phase, we made an assumption that the removal of newline characters and tabs would not substantially affect the model's code generation capabilities. This assumption was based on the idea that these characters, while useful for formatting, might not contribute to the semantic content that the model uses to understand the code structure and purpose. To test this, we conducted controlled experiments in Table 8, where we systematically removed these characters from the DocStrings and compared the model's Pass@1 score against a control group where these elements were retained.

Impact of Stop Words. We also investigated the role of stop words in DocStrings. In our study, we define the stop words are commonly considered to be filler words that do not carry much meaning, which are chosen by LLMLingua2 used in a 10% ratio we discussed in Section 3. Our experiments

Table 7. Pass@1 Comparison within Different Method Name Quality

Dataset	Method	LLMs					
		DS-1.3B	DS-6.7B	CQ-7.3B	CG-9.4B	LA-8.0B	Avg.
HumanEval	Original	63.42	71.95	77.44	60.37	57.32	66.10
	ShortenDoc	**57.93**	**72.56**	**78.66**	**64.34**	**56.10**	**65.92**
	ShortenDoc w Foo	50.61	71.95	68.90	54.27	53.05	59.96
CodeHarmony	Original	59.48	64.36	60.78	64.71	59.48	61.76
	ShortenDoc	**55.56**	**66.01**	**62.75**	**60.78**	**61.44**	**61.31**
	ShortenDoc w Foo	48.37	60.13	58.82	51.63	54.25	54.64
MBPP	Original	36.40	46.20	59.00	46.60	41.40	45.92
	ShortenDoc	**36.60**	**43.80**	**55.40**	**46.20**	**41.40**	**44.68**
	ShortenDoc w Foo	30.80	38.60	47.20	37.40	37.20	38.24
Subtle	Original	53.00	61.00	63.00	64.00	56.00	59.40
	ShortenDoc	**54.00**	**56.00**	**62.00**	**66.00**	**56.00**	**58.80**
	ShortenDoc w Foo	39.00	48.00	58.00	53.00	54.00	50.40
Creative	Original	23.00	34.00	38.00	36.00	34.00	33.00
	ShortenDoc	**23.00**	**36.00**	**37.00**	**35.00**	**35.00**	**33.20**
	ShortenDoc w Foo	22.00	33.00	39.00	36.00	25.00	31.00
BigCodeBench	Original	6.10	12.20	13.50	14.20	10.80	11.36
	ShortenDoc	2.00	6.10	10.10	9.50	4.70	6.48
	ShortenDoc w Rename	**2.00**	**7.40**	**10.80**	**12.20**	**8.10**	**8.10**

Bold means the best performance in the comparison experiment.

in Table 9 involved removing stop words from DocStrings to assess if the model's Pass@1 score would change. The comparison between the Pass@1 score with and without stop words helped us understand the contribution of these words to the overall comprehension of different LLMs.

Impact of Redundant Instruction. Our analysis of the MBPP dataset revealed a prevalent pattern of redundant instructions, specifically the phrase "write a function to." Intuitively, we suspected that such repetitive and generic instructions might not be beneficial for the model in understanding the specific requirements of the code it is tasked to generate. To analyze the impact of these instructions, we designed a set of controlled experiments where we removed these phrases from the DocStrings. The comparison of the model's output with and without these redundant instructions allowed us to evaluate their utility in the context of code generation.

Findings and Implications. The findings from these experiments shed light on the importance of various elements within DocStrings. The removal of newline characters and tabs showed varying impacts across different datasets. While most datasets maintained stable performance, we observed a notable decline in BigCodeBench dataset (from 11.36% to 7.32%). This degradation could be attributed to two main reasons: (1) the tokenization mechanism where models may process `"""\n` differently from `"""` alone, potentially affecting the model's ability to properly parse the DocString; and (2) the inherent complexity of BigCodeBench dataset, which presents significant challenges for LLMs, making them more sensitive to any modifications in the input format.

The impact of stop words was also nuanced, with some instances showing a slight Pass@1 change, indicating that even seemingly inconsequential words might play a role in the model's

Table 8. Pass@1 Comparison within Different DocString Styles

Dataset	Method	LLMs					
		DS-1.3B	DS-6.7B	CQ-7.3B	CG-9.4B	LA-8.0B	Avg.
HumanEval	Original	63.42	71.95	77.44	60.37	57.32	66.10
	Remove New Line and Tab	57.32	76.83	83.84	66.46	60.37	68.96
	Remove Stop Words	54.88	66.46	76.83	56.10	56.10	62.07
CodeHarmony	Original	59.48	64.36	60.78	64.71	59.48	61.76
	Remove New Line and Tab	60.78	67.32	62.75	61.44	61.44	62.75
	Remove Stop Words	57.52	67.32	60.14	60.78	58.17	60.79
MBPP	Original	36.40	46.20	59.00	46.60	41.40	45.92
	Remove New Line and Tab	37.80	47.20	59.00	47.80	43.00	46.96
	Remove Stop Words	34.00	45.80	54.40	44.20	41.20	43.92
Subtle	Original	53.00	61.00	63.00	64.00	56.00	59.40
	Remove New Line and Tab	54.00	58.00	62.00	68.00	60.00	60.40
	Remove Stop Words	46.00	56.00	61.00	62.00	55.00	56.00
Creative	Original	23.00	34.00	38.00	36.00	34.00	33.00
	Remove New Line and Tab	22.00	31.00	33.00	34.00	30.00	30.00
	Remove Stop Words	20.00	29.00	29.00	30.00	31.00	27.80
BigCodeBench	Original	6.10	12.20	13.50	14.20	10.80	11.36
	Remove New Line and Tab	1.40	8.80	12.20	7.40	6.80	7.32
	Remove Stop Words	2.0	8.10	9.50	10.80	6.10	7.30

Table 9. Pass@1 Comparison of Redundant Instruction

Dataset	Method	LLMs					
		DS-1.3B	DS-6.7B	CQ-7.3B	CG-9.4B	LA-8.0B	Avg.
MBPP	Original	36.40	46.20	59.00	46.60	41.40	45.92
	w/o "Write a python function to"	37.80	45.60	57.40	47.60	42.80	46.24

comprehension process. The removal of redundant instructions led to a modest improvement in Pass@1 score, supporting our hypothesis that such phrases do not contribute positively to the model's understanding and may even introduce noise.

These experiments underscore the importance of a balanced approach to DocString style. While it is beneficial to maintain clear and concise documentation, the removal of certain elements should be done with consideration of their potential impact on the model's comprehension.

6.3 Hyper-Parameter Analysis

(1) The hyper-parameter τ significantly influences the performance of our method. Since the hyper-parameter τ is used to determine the threshold for token importance, it directly affects the number of tokens retained in the compressed DocString, i.e., the compression ratio. As demonstrated in Table 10, different values of τ result in varying performance metrics across the CodeHarmony dataset. The optimal value of τ that maximizes performance may differ depending on the specific dataset and model architecture. For instance, a higher τ value such as 0.999 seems to work better for the CodeQwen, CodeGeeX4, and Llama3.1 model, while a lower value like 0.985 could be more

Table 10. Pass@1 Comparison of Different τ in CodeHarmony

Dataset	τ	Ratio	LLMs					
			DS-1.3B	DS-6.7B	CQ-7.3B	CG-9.4B	LA-8.0B	Avg.
CodeHarmony	-	0	59.48	64.36	60.78	64.71	59.48	61.76
	0.985	32	**58.82**	62.09	59.48	56.86	56.21	58.69
	0.990	29	56.21	64.05	62.10	59.48	58.17	60.00
	0.995	26	56.86	**66.67**	62.10	59.48	60.78	61.18
	0.999	25	56.21	66.01	**63.40**	**60.78**	**61.44**	**61.44**

Bold means the best performance in the comparison experiment.

Table 11. Pass@1 Comparison of Different Base Models in CodeHarmony

Dataset	Base Model	LLMs					
		DS-1.3B	DS-6.7B	CQ-7.3B	CG-9.4B	LA-8.0B	Avg.
CodeHarmony	CodeGen-350M	52.94	62.75	57.52	54.25	52.94	56.08
	GPT2	41.83	45.75	47.71	43.14	44.44	44.57
	CodeGPT-py-adapted	**56.21**	**66.01**	**63.40**	**60.78**	**61.44**	**61.44**

Bold means the best performance in the comparison experiment.

effective for the DeepSeekCoder-1.3B model. Given the good balance or optimal Pass@1 score for the models and datasets, we set the hyper-parameter τ to 0.999.

A limitation in our hyper-parameter analysis is that, while we conducted detailed τ experiments on CodeHarmony (as shown in Table 10), we did not perform exhaustive τ tuning across all datasets. The optimal τ value might vary across datasets due to their respective characteristics, and our choice of $\tau = 0.999$ based on CodeHarmony may not be optimal for all datasets. Nevertheless, the experimental results demonstrate that $\tau = 0.999$ consistently provides competitive Pass@1 scores across different datasets, as shown in Table 3. For instance, ShortenDoc achieves comparable or better Pass@1 than baseline methods in HumanEval (78.66%), CodeHarmony (63.40%), and MBPP (55.40%) when using the CodeQwen model.

We leave as future work a more comprehensive analysis of τ across different datasets to better understand its impact on compression performance and to develop dataset-specific tuning strategies.

(2) The choice of base model plays a critical role in the performance of our method. We find that different base models can lead to varying token importance rankings, indicating that the token sorting decisions are not entirely consistent across models. This observation raises interesting questions about the relationship between model characteristics and their ability to identify important tokens in DocStrings.

As shown in Table 11, different base models demonstrate varying baseline performances which can be attributed to their pre-training strategies: CodeGen-350M pre-trained specifically on code corpora outperforms GPT2 pre-trained solely on natural language corpora. Notably, CodeGPT-py-adapted achieves the best performance as it combines both approaches; it is built upon GPT2 and further pre-trained on code corpora. This suggests that the ability to understand both natural language and code context is crucial for effective DocString compression.

For the CodeHarmony dataset, while the CodeGPT-py-adapted base model outperforms the other models across all metrics, the token importance rankings produced by different models show notable variations. Despite these differences in token sorting, our method maintains robust

performance across different base models, suggesting that the overall compression framework is resilient to variations in token importance assessment.

In summary, we advise users to set 0.999 as the default value of τ for balanced performance. If users wish to experiment with different compression ratios, they may gradually decrease this value. Meanwhile, we recommend using the CodeGPT-py-adapted model as the base model for DocString compression, as it consistently achieves the best performance across different datasets.

6.4 Threats to Validity

In this subsection, we analyze potential threats to the validity of our empirical study.

Threats to Internal Validity. The first internal threat is the possibility of implementation errors in ShortenDoc. To alleviate this, we conducted a thorough code inspection of the implementation and utilized mature libraries. The second internal threat is the implementation correctness of the considered baselines. To mitigate this threat, we implemented all baselines by running their open source code directly or reimplementing them according to the original studies.

Threats to External Validity. The main external threat lies in the choice of datasets and models used in our study. For Python code generation, we selected six diverse datasets with high reputations to reflect the complexity of real-world scenarios. However, for cross-language evaluation, we relied on HumanEval-X, which consists of translations of the original HumanEval dataset. The reliance on translated versions of a single dataset, rather than naturally occurring code in different languages, might limit the generalizability of our findings. While this choice was constrained by the current availability of multi-lingual code generation benchmarks, future work could benefit from evaluating on native, language-specific datasets as they become available.

In terms of the choice of models, we prioritized open source models for better reproducibility and community verification, selecting six representative models that span different scales (from 1.3B to 9.4B parameters) and architectures. These include specialized code models (DeepSeekCoder-1.3b, DeepSeekCoder-6.7b, CodeQwen1.5, CodeGeeX4) and general-purpose models (Llama3.1). While proprietary models such as GPT-4 might offer superior performance, their high API costs and frequent version updates could impact the stability and reproducibility of our results. Given the hardware constraint (single 3090 GPU) and the need for experimental consistency, we believe our current model selection provides meaningful insights into the effectiveness of our compression method across different scenarios.

Threats to Construct Validity. Construct threats concern the performance metrics used to evaluate ShortenDoc and baselines. To evaluate the performance of models, we utilized Pass@1 metric which is commonly used in in the previous studies of code generation. Additionally, we used Compression Ratio, which is also widely used in similar studies, to measure the proportion of reduction in the length of the prompt after compression.

Furthermore, we conducted a human study to analyze the qualitative impact of DocString compression, where the Informativeness and Comprehensibility metrics are used which might not capture all aspects of the DocString quality. We recruited evaluators with strong Python programming backgrounds and provided them with clear evaluation criteria. Additionally, we implemented measures to ensure evaluation quality, including anonymized compressed DocStrings, allowing evaluators to research unfamiliar concepts, and limiting their work load.

7 Related Work

7.1 Code Generation

Code generation aims to produce code snippets from given natural language descriptions or requirements. Some studies [22, 28] use sequence-based models, which treat the source code as

a sequence of tokens and utilize neural networks to generate the source code token by token based on the input description. Other studies [46, 56] use tree-based models, i.e., construct a parse tree of the program from the natural language description and subsequently convert it into corresponding code.

In recent years, researchers have gradually utilized pre-trained models for code generation tasks, which have outperformed conventional sequence-based and tree-based methods. These models are pre-trained on massive data of source code and then fine-tuned on code generation task. For example, models like CodeGPT [24], PLBART [1], and CodeT5 [50] leverage the GPT, BART, and T5 architectures of language models pre-trained on code corpora. However, these models are more suitable for fine-tuning code generation tasks, as their parameter numbers are not large enough to demonstrate emergent capabilities in zero-shot scenarios.

With the development of LLM research, LLMs with over a billion parameters have been employed for zero-shot code generation tasks. Current Code LLMs can be divided into two categories: standard language models and instruction-tuned models [18].

Standard language models are pre-trained on the raw corpus with the next-token prediction. With the success of GPT series [4, 38, 39] in NLP, Chen et al. [6] adapted similar ideas into the domain of source code and fine-tunes GPT models on code to produce closed-source CodeX, which is pre-trained on GitHub code with 12 billion model parameters. To replicate its success, Nijkamp et al. [32] proposed CodeGen and CodeGen2, which are LLMs for code with multi-turn program synthesis. Zheng et al. [57] proposed CodeGeeX, a large-scale multi-lingual code generation model with 13 billion parameters. CodeGeeX is pre-trained on a large code corpus of over 20 programming languages and has good performance for generating executable programs in several mainstream programming languages like Python, C++, Java, JavaScript, Go, and so on. Li et al. [19] proposed StarCoder, a 15.5 billion parameter model whose training data incorporates more than 80 different programming languages as well as text extracted from GitHub issues and commits and from notebooks. Differing from the aforementioned decoder-only model, Wang et al. [49] proposed CodeT5+, a family of encoder-decoder LLMs for code in which component modules can be flexibly combined to suit a wide range of downstream code tasks.

Instruction-tuned models are fine-tuned using instruction tuning [51]. Instruction tuning helps models follow user's instructions. OpenAI's ChatGPT [34] is trained by Reinforcement Learning with Human Feedback [35], making it capable of programming tasks. However, it is of closed-source. For the open source models, Luo et al. [25] introduce Wizardcoder by fine-tuning StarCoder [19] with Evol-Instruct and ChatGPT feedback with Code Alpaca's dataset as seed dataset. Wang et al. introduce InstructCodeT5+ [49] by fine-tuning CodeT5+ [49] on Code Alpaca's [5] dataset.

In this article, our main goal is to compress DocStrings within code prompts without losing their semantic integrity in the field of code generation. We have observed that the model can still understand the task requirements and generate correct code after removing some of the redundant information in DocStrings. This observation has motivated us to further investigate the compression of DocStrings, an essential component of prompt. To achieve the goal, we have conducted an empirical study on code generation task.

7.2 Prompt Compression

Prompt compression attempts to shorten the original prompts without losing essential information. Prompt compression methods [16] can be grouped into three main categories: token pruning and token merging, soft prompt tuning methods, and information-entropy-based approaches. Token pruning and token merging need model fine-tuning or intermediate results during inference and have been used with BERT-scale models. For instance, Modarressi et al. [27] introduced AdapLeR which dynamically eliminates less contributing tokens through layers to achieve shorter lengths.

Soft prompt tuning requires LLMs' parameter fine-tuning to make them suitable for specific domains but not applicable to black-box LLMs. Wingate et al. [52] used the framework of soft prompts to manipulate prompt compression. Mu et al. [29] proposed GIST, a manner to compress arbitrary prompts into a smaller set of Transformer activations on top of virtual "gist" tokens. However, these methods are task-aware and usually tailored for specific tasks and compression ratios, which may limit their generalizability in real-world scenarios.

Information-entropy-based approaches use a small language model to calculate the self-information or perplexity of each token in the original prompt and then remove tokens with lower perplexities. For example, Li et al. [21] introduced Selective_Context, which employs self-information to filter out less informative content, resulting in the efficiency of the fixed context length. Jiang et al. [15] proposed LLMLingua, a coarse-to-fine prompt compression method to reduce the length of original prompts while preserving essential information. Based on LLMLingua, Pan et al. [36] proposed a data distillation procedure to derive knowledge from an LLM (GPT-4) to compress the prompts without losing crucial information. All of these approaches are task-agnostic prompt compression methods and have better generalizability and efficiency compared with task-aware methods.

In contrast to the previous studies, we present a novel adaptive compression approach targeting code generation. This compression approach emphasizes the understanding of code semantics and removes redundant content by analyzing the importance of individual tokens. To improve the efficiency of prompt compression, we employ a Top-N strategy, which optimize the compression process and greatly preserve the semantic integrity.

8 Conclusion and Future Work

In our study, we focus on DocString compression and avoiding the loss of essential information in DocString. Thereby, we propose a novel compression method ShortenDoc. This compression method dynamically adjusts the compression rate and retains greater informativeness and comprehensibility in compressed DocStrings. By implementing this compression method, our goal is to improve model efficiency and reduce the computational resources cost.

While our current research has focused primarily on function level code generation, we recognize that the complexity of code generation increases as the code structure expands. Therefore, future work will extend to class level code generation, which will involve more complex logic and larger code structures. We believe that ShortenDoc's approach can accommodate these higher-level code structures while maintaining its compression efficiency and output quality.

In addition, we plan to explore code generation in a multi-language environment. With the diversification of global software development, DocString compression tools that support multiple programming languages will be of higher utility. We plan to train a multi-lingual base language model, which will enhance the applicability and performance of ShortenDoc in different programming languages.

Furthermore, we aim to extend our research to general code completion tasks. Given the higher frequency of code completion in real-world software engineering compared to natural language to code generation, the motivation for prompt compression becomes even stronger in this context. This extension could potentially bring more significant practical benefits in terms of both computational and economic efficiency.

References

[1] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. arXiv:2103.06333. Retrieved from https://arxiv.org/abs/2103.06333

[2] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. arXiv:2108.07732. Retrieved from https://arxiv.org/abs/2108.07732

[3] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. 2023. Qwen technical report. arXiv:2309.16609. Retrieved from https://arxiv.org/abs/2309.16609

[4] Tom B. Brown. 2020. Language models are few-shot learners. arXiv:2005.14165. Retrieved from https://arxiv.org/abs/2005.14165

[5] Sahil Chaudhary. 2023. Code alpaca: An instruction-following Llama model for code generation. *GitHub Repository*. Retrieved from https://github.com/sahil280114/codealpaca

[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. arXiv:2107.03374. Retrieved from https://arxiv.org/abs/2107.03374

[7] Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. 2024. ChatUniTest: A framework for LLM-based test generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, 572–576.

[8] Nicola Dainese, Alexander Ilin, and Pekka Marttinen. 2024. Can DocString reformulation with an LLM improve code generation? In *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics: Student Research Workshop*, 296–312.

[9] Xi Ding, Rui Peng, Xiangping Chen, Yuan Huang, Jing Bian, and Zibin Zheng. 2024. Do code summarization models process too much information? Function signature may be all that is needed. *ACM Transactions on Software Engineering and Methodology* 33, 6 (2024), 1–35.

[10] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The Llama 3 herd of models. arXiv:2407.21783. Retrieved from https://arxiv.org/abs/2407.21783

[11] Shubhang Shekhar Dvivedi, Vyshnav Vijay, Sai Leela Rahul Pujari, Shoumik Lodh, and Dhruv Kumar. 2024. A comparative analysis of large language models for code documentation generation. In *Proceedings of the 1st ACM International Conference on AI-Powered Software*, 65–73.

[12] Flab-Pruner. 2024. Flab-Pruner: Towards Greener yet Powerful Code Intelligence via Structural Pruning. Retrieved from https://github.com/Flab-Pruner/Flab-Pruner

[13] Joseph L. Fleiss. 1971. Measuring nominal scale agreement among many raters. *Psychological Bulletin* 76, 5 (1971), 378–382.

[14] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, et al. 2024. DeepSeek-Coder: When the large language model meets programming–The rise of code intelligence. arXiv:2401.14196. Retrieved from https://arxiv.org/abs/2401.14196

[15] Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. 2023. LLMLingua: Compressing prompts for accelerated inference of large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 13358–13376.

[16] Huiqiang Jiang, Qianhui Wu, Xufang Luo, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. 2023. LongLLMLingua: Accelerating and enhancing LLMs in long context scenarios via prompt compression. arXiv:2310.06839. Retrieved from https://arxiv.org/abs/2310.06839

[17] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

[18] Jia Li, Yunfei Zhao, Yongmin Li, Ge Li, and Zhi Jin. 2024. AceCoder: An effective prompting technique specialized in code generation. *ACM Transactions on Software Engineering and Methodology* 33, 8 (2024), 1–26.

[19] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. Starcoder: May the source be with you! arXiv:2305.06161. Retrieved from https://arxiv.org/abs/2305.06161

[20] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with AlphaCode. *Science* 378, 6624 (2022), 1092–1097.

[21] Yucheng Li, Bo Dong, Frank Guerin, and Chenghua Lin. 2023. Compressing context to enhance inference efficiency of large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 6342–6353.

[22] Wang Ling, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiskỳ, Andrew Senior, Fumin Wang, and Phil Blunsom. 2016. Latent predictor networks for code generation. arXiv:1603.06744. Retrieved from https://arxiv.org/abs/1603.06744

[23] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024. Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation. In *Proceedings of the Advances in Neural Information Processing Systems*, Vol. 36.

[24] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. In *Proceedings of the 35th Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*.

[25] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with Evol-Instruct. arXiv:2306.08568. Retrieved from https://arxiv.org/abs/2306.08568

[26] Antonio Valerio Miceli-Barone and Rico Sennrich. 2017. A parallel corpus of Python functions and documentation strings for automated code documentation and code generation. In *Proceedings of the 8th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, 314–319.

[27] Ali Modarressi, Hosein Mohebbi, and Mohammad Taher Pilehvar. 2022. AdapLeR: Speeding up inference by adaptive length reduction. arXiv:2203.08991. Retrieved from https://arxiv.org/abs/2203.08991

[28] Lili Mou, Rui Men, Ge Li, Lu Zhang, and Zhi Jin. 2015. On end-to-end program generation from user intention by deep neural networks. arXiv:1510.07211. Retrieved from https://arxiv.org/abs/1510.07211

[29] Jesse Mu, Xiang Li, and Noah Goodman. 2024. Learning to compress prompts with gist tokens. In *Proceedings of the Advances in Neural Information Processing Systems*, Vol. 36.

[30] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2024. Using an LLM to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 1–13.

[31] Arvind Neelakantan, Tao Xu, Raul Puri, Alec Radford, Jesse Michael Han, Jerry Tworek, Qiming Yuan, Nikolas Tezak, Jong Wook Kim, Chris Hallacy, et al. 2022. Text and code embeddings by contrastive pre-training. arXiv:2201.10005. Retrieved from https://arxiv.org/abs/2201.10005

[32] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. CodeGen: An open large language model for code with multi-turn program synthesis. arXiv:2203.13474. Retrieved from https://arxiv.org/abs/2203.13474

[33] Changan Niu, Ting Zhang, Chuanyi Li, Bin Luo, and Vincent Ng. 2024. On evaluating the efficiency of source code generated by LLMs. In *Proceedings of the 2024 IEEE/ACM 1st International Conference on AI Foundation Models and Software Engineering*, 103–107.

[34] OpenAI. 2022. ChatGPT. Retrieved from https://openai.com/blog/chatgpt

[35] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. In *Proceedings of the Advances in Neural Information Processing Systems*, Vol. 35, 27730–27744.

[36] Zhuoshi Pan, Qianhui Wu, Huiqiang Jiang, Menglin Xia, Xufang Luo, Jue Zhang, Qingwei Lin, Victor Rühle, Yuqing Yang, Chin-Yew Lin, et al. 2024. LLMLingua-2: Data distillation for efficient and faithful task-agnostic prompt compression. In *Findings of the Association for Computational Linguistics (ACL '24)*. Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.), Association for Computational Linguistics, Bangkok, Thailand, 963–981. Retrieved from https://aclanthology.org/2024.findings-acl.57

[37] Bibek Poudel, Adam Cook, Sekou Traore, and Shelah Ameli. 2024. DocuMint: Docstring generation for Python using small language models. arXiv:2405.10243. Retrieved from https://arxiv.org/abs/2405.10243

[38] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. Improving language understanding by generative pre-training. OpenAI Blog.

[39] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners. *OpenAI Blog* 1, 8 (2019), 9.

[40] Justus J. Randolph. 2005. Free-Marginal Multirater Kappa (Multirater K [Free]): An Alternative to Fleiss' Fixed-Marginal Multirater Kappa. In *Presented at the Joensuu Learning and Instruction Symposium*.

[41] Sanka Rasnayaka, Guanlin Wang, Ridwan Shariffdeen, and Ganesh Neelakanta Iyer. 2024. An empirical study on usage and perceptions of LLMs in a software engineering project. In *Proceedings of the 1st International Workshop on Large Language Models for Code*, 111–118.

[42] Jieke Shi, Zhou Yang, Hong Jin Kang, Bowen Xu, Junda He, and David Lo. 2024. Greening large language models of code. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Society*, 142–153.

[43] Zhensu Sun, Xiaoning Du, Fu Song, Shangwen Wang, and Li Li. 2024. When neural code completion models size up the situation: Attaining cheaper and faster completion through dynamic model inference. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 1–12.

[44] Zhensu Sun, Xiaoning Du, Fu Song, Shangwen Wang, Mingze Ni, Li Li, and David Lo. 2024. Don't complete it! Preventing unhelpful code completion for productive and sustainable neural code completion systems. *ACM Transactions on Software Engineering and Methodology* 34, 1 (2024), 1–22.

[45] Zhensu Sun, Xiaoning Du, Zhou Yang, Li Li, and David Lo. 2024. AI coders are among us: Rethinking programming language grammar towards efficient code generation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 1124–1136.

[46] Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang. 2020. TreeGen: A tree-based transformer architecture for code generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34, 8984–8991.

[47] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017. Isabelle Guyon, Ulrike Von Luxburg*, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.), 5998–6008. Retrieved from https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[48] Shiqi Wang, Zheng Li, Haifeng Qian, Chenghao Yang, Zijian Wang, Mingyue Shang, Varun Kumar, Samson Tan, Baishakhi Ray, Parminder Bhatia, et al.2023. ReCode: Robustness evaluation of code generation models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics*.

[49] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi.2023. Codet5+: Open code large language models for code understanding and generation. arXiv:2305.07922. Retrieved from https://arxiv.org/abs/2305.07922

[50] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. arXiv:2109.00859. Retrieved from https://arxiv.org/abs/2109.00859

[51] Jason Wei, Maarten Bosma, Vincent Y. Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V. Le. 2021. Finetuned language models are zero-shot learners. arXiv:2109.01652. Retrieved from https://arxiv.org/abs/2109.01652

[52] David Wingate, Mohammad Shoeybi, and Taylor Sorensen. 2022. Prompt compression and contrastive conditioning for controllability and toxicity reduction in language models. arXiv:2210.03162. Retrieved from https://arxiv.org/abs/2210.03162

[53] Chunqiu Steven Xia, Yinlin Deng, and Lingming Zhang. 2024. Top leaderboard ranking= top coding proficiency, always? EvoEval: Evolving coding benchmarks via LLM. arXiv:2403.19114. Retrieved from https://arxiv.org/abs/2403.19114

[54] Guang Yang, Yu Zhou, Wenhua Yang, Tao Yue, Xiang Chen, and Taolue Chen. 2024. How important are good method names in neural code generation? A model robustness perspective. *ACM Transactions on Software Engineering and Methodology* 33, 3 (2024), 1–35.

[55] Zhou Yang, Zhensu Sun, Terry Zhuo Yue, Premkumar Devanbu, and David Lo. 2024. Robustness, security, privacy, explainability, efficiency, and usability of large language models for code. arXiv:2403.07506. Retrieved from https://arxiv.org/abs/2403.07506

[56] Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. arXiv:1704.01696. Retrieved from https://arxiv.org/abs/1704.01696

[57] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, et al. 2023. CodeGeeX: A pre-trained model for code generation with multilingual benchmarking on HumanEval-X. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 5673–5684.

[58] Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. 2024. BigCodeBench: Benchmarking code generation with diverse function calls and complex instructions. arXiv:2406.15877. Retrieved from https://arxiv.org/abs/2406.15877

Received 30 October 2024; revised 24 April 2025; accepted 6 May 2025