Free your mouse! Command Large Language Models to Generate Code to Format Word Documents

Shihao Rao[1,2], **Liang Li**[1]*, **Jiapeng Liu**[1,2], **Weinxin Guan**[1,2],
Xiyan Gao[1], **Bing Li**[1], **Can Ma**[1]
[1]Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
[2]School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China
{raoshihao, liliang}@iie.ac.cn

Abstract

Recently, LLMs have significantly improved code generation, making it increasingly accessible to users. As a result, LLM-powered code generation applications have sprung up, vastly boosting user productivity. This paper mainly explores how to improve the efficiency and experience of users in formatting the document. Specifically, we propose an automatic document formatting method, TEXT-TO-FORMAT, which is driven by various prompting strategies. TEXT-TO-FORMAT takes the user's formatting instructions and then generates code that can be run in Microsoft Word to format the content in a document. Further, to evaluate automatic document formatting approaches and advance the document formatting task, we build an evaluation specification including a high-quality dataset DOCFORMEVAL, a code runtime environment, and evaluation metrics. Extensive experimental results on DOCFORMEVAL reveal that the prompting strategy's effect positively correlates with how much knowledge it introduces related to document formatting task. We believe the constructed DOCFORMEVAL and the exploration about TEXT-TO-FORMAT can help developers build more intelligent tools for automatic document formatting, especially in offline scenarios, where the data privacy is the top priority [1].

1 Introduction

Code generation (Zheng et al., 2023; Liu et al., 2024; Jiang et al., 2024) aims to transform task requirements expressed in natural language into executable code. It improves the productivity of both non-expert users and developers. For example, text-to-SQL (Yu et al., 2018) enhances development efficiency for programmers while lowering the threshold for end-users to access the knowledge stored in the database. Recently, the development

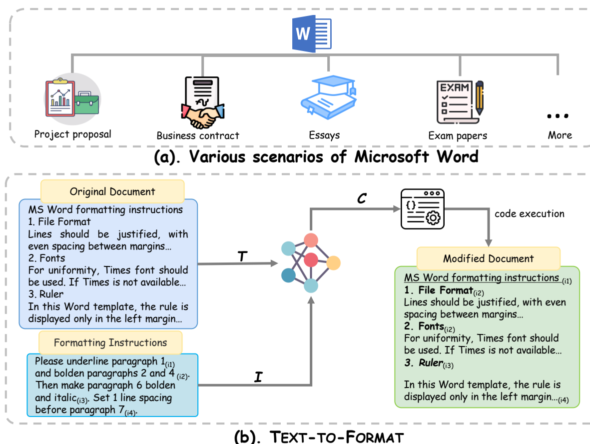

Figure 1: Diagram of the document formatting task and the automated formatting method based on code generation.

of Large Language Models (LLMs) (Zhang et al., 2023b; Zhuo, 2023; Coignion et al., 2024; Tang et al., 2024) has led to the innovation of the code generation domain. As a result, LLM-powered code generation applications have sprung up, e.g., Copilit (Dakhel et al., 2023) and OpenDevin (Team, 2024), vastly boosting user productivity and experience. What other scenarios can benefit from code generation deserves to be explored.

Microsoft Word, a globally popular word processing software, has a set of rich and powerful tools for document creation and editing. As depicted in Figure 1 (a), it is utilized by individuals from various sectors, such as business and education, to compose project proposals, draft business contracts, write essays, make test papers, etc. After completing content creation, it is necessary to set the document format, which varies from scenario to scenario. We refer to the document formatting task as DOCFORM. To achieve this, Microsoft Word enables users to adjust document format using the mouse to select and click, including changing fonts, sizes, colors, spacing, etc. However, when formatting documents, especially complex ones, users

*Corresponding author: Liang Li
[1]Our code and data are released publicly: https://github.com/RaoShiHao/doc-formatting

Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, pages 16127–16142
November 12-16, 2024 ©2024 Association for Computational Linguistics

often encounter numerous repetitive formatting operations, as well as the necessary format validation work, which can take up the user's creative time. To minimize formatting effort, previous work (Konglong, 2023) provides limited auto-formatting functions for specific requirements through preset templates. On the one hand, the increasing number of formatting requirements means that experts have to draft more templates, which is time-consuming and costly manually. On the other hand, the formatting standards vary according to the users' needs, making it almost impossible to address all possible variations.

To free users from tedious and repetitive formatting work and allow them to focus on more challenging content creation, developing a more intelligent automated document formatting tool is imperative and valuable. Inspired by the recent success of code generation, in this paper, we propose TEXT-TO-FORMAT, an automatic document formatting method powered by LLMs. Specifically, as illustrated in Figure 1 (b), TEXT-TO-FORMAT takes in natural language formatting instructions I provided by the user and generates a formatting code C for the specified text T. Upon executing the code C, the format of the specified text can be automatically adjusted as the user desires.

It is noticed that existing code generation evaluation datasets, e.g., HumanEval (Chen et al., 2021) and DS-1000 (Lai et al., 2022), lack test instructions for the document formatting task. Therefore, to better validate the feasibility of TEXT-TO-FORMAT, we first construct an evaluation dataset, DOCFORMEVAL, for the document formatting task. We make three efforts to improve the quality of DOCFORMEVAL and guarantee it is close to practical document formatting scenarios. First, we delve deep into Microsoft Word to gather 125 atomic formatting operations on font and paragraph levels and heuristically synthesize the initial input-output data D^s. Then, inspired by recent advancements in data generation (Wang et al., 2022c; Du et al., 2023; Yu et al., 2024), we utilize the powerful GPT-4 Turbo to diversify the text formatting instructions in D^s to align with human language representations. Lastly, we manually review the rewritten data and discard low-quality examples.

As LLMs-driven methods have dominated various code generation tasks, in this paper, we focus on exploring the impact of different prompt strategies on TEXT-TO-FORMAT performance. Specifically, we first compare the performance of two methods, prompt learning (Madaan et al., 2022; Liu et al., 2023a; Nong et al., 2024), and retrieval-augmented generation (RAG) (Zan et al., 2022; Wu et al., 2023; Su et al., 2024), on DOCFORMEVAL in the zero-shot setting. Furthermore, the impact of the number of demonstrations used in the prompting on the results is explored in the few-shot setting. Finally, we explore the impact of the self-refinement mechanism (Chen et al., 2023; Madaan et al., 2024) on automatic document formatting.

Our contributions are three-fold as follows:

- We highlight a valuable application scenario for code generation, i.e., document formatting in Microsoft Word. To advance it, we build an evaluation specification including a high-quality dataset DOCFORMEVAL, a code runtime environment, and evaluation metrics.
- We propose TEXT-TO-FORMAT, which formats text by transforming user formatting instructions into formatting code. Moreover, we have investigated the impact of various prompting strategies on TEXT-TO-FORMAT.
- The constructed DOCFORMEVAL and the exploration of various TEXT-TO-FORMAT methods can assist developers in building more intelligent tools for automatic document formatting, **especially in offline scenarios**, where the data privacy is the top priority.

2 Related Work

In-Context Learning for Code Generation Recently, Large Language Models (LLMs) (Roziere et al., 2023; Achiam et al., 2023; DeepSeek-AI, 2024) have showcased remarkable knowledge transfer and logical reasoning abilities in few-shot or zero-shot scenarios (Madaan et al., 2022). These capabilities attract many works that focus on exploring how to provide well-designed in-context examples for the current prompt, guiding LLMs to generate compliant code more effectively. studies (Wang et al., 2022b) attempt to add examples of input-output directly into the context. On the other hand, studies (Chen et al., 2023; Wu et al., 2023; Liu et al., 2024) are inspired by the specialty of code generation itself, adding tutorials on how to self-debug code to in-context examples, thereby prompting LLMs to generate more robust code.

Retrieval-Augmented Code Generation LLMs typically have difficulty covering all types of code due to their inherently outdated knowledge. Ad-

ditionally, the high frequency of code updates and iterations makes it costly to continually fine-tune LLMs based on new information. Therefore, some research efforts are leveraging the concept of retrieval-augmented generation (RAG) (Shi et al., 2023; Xu et al., 2023; Jiang et al., 2023b), where external knowledge relevant to the current prompt is provided to the LLMs as additional auxiliary cues. For instance, Doc Prompting (Wu et al., 2023) utilizes a natural language to code generation method by retrieving code documentation. In addition, APIRetriever (Zan et al., 2022) introduces a framework designed to adapt LLMs to private libraries, leveraging an APICoder to generate code using these API docs. Meanwhile, RepoCoder (Zhang et al., 2023a) employs the iterative generate-retrieval procedure to do repository-level code completion. Inspired by those works, this paper introduces RAG to improve the TEXT-TO-FORMAT.

3 DocFormEval Dataset

3.1 Problem Formulation

We formally define the document formatting task (DocForm) driven by code generation as follows. As illustrated in Figure 1 (b), the input is a natural language instruction I that contains multiple formatting requirements. Given the instruction I, the objective of DocForm is to generate a code snippet $C = \{c_1, c_2, ..., c_n\}$ that can be run in the Microsoft Word environment $\mathcal{E}$ to format specified content T in a document:

$$C = \operatorname{argmax} \prod_{i=1}^{n} P(c_i | c_{<i}; \theta) \tag{1}$$

where θ denotes the parameters of a neural text generation model, and c_i denotes the i-th tokens in the generated code snippet.

3.2 DocFormEval Construction Pipeline

It is observed that each document formatting instruction I involves several atomic formatting operations $O = \{o_1, o_2, ..., o_m\}$, where o_i denotes an atomic operation. In Microsoft Word, an atomic operation only formats a property of a font or paragraph. Inspired by this, we combine different atomic operations to obtain the backbone O of each formatting instruction, which is then rewritten into natural language instructions to construct our evaluation dataset DocFormEval. To ensure the dataset quality, we carefully design its construction pipeline as following steps:

Atomic Operation Collection We first define each atomic operation as a 4-tuple $o = (k, v, s, d)$, where k is the property of the formatting operation, v is the property's value, s is a manually written code fragment that implements the operation, and d denotes its complexity. d is determined by the number of lines in s. For example, the atomic operation `"set the font size to 10"` is represented as :

```
o = {
k: "font-size",
v: "10 pt",
s: "font.size = 10;",
d: 1 }
```

Then, we thoroughly investigate the API documentation[2] provided by Microsoft. Based on this, as displayed in Table 4, we choose 19 properties in the API documentation that could correspond one-to-one with the Word function bar. Furthermore, to prevent excessive testing for a specific property, we randomly select $4 \sim 12$ values from its optional range for each property. Lastly, by combining attributes and their sampled values, we obtain a set O^a containing 125 atomic formatting operations at the font and paragraph levels.

Complex Operation Combination As previously mentioned, document formatting instructions can be complex and usually contain multiple atomic operations. To simulate more complex formatting requirements from users, we randomly combine N atomic operations from the set O^a to form the backbone of complex formatting instructions. Specifically, we follow the following principles in combing: (1) Do not consider atomic operations belonging to the four properties `superscript`, `subscript`, `strikethrough`, and `double strikethrough`. In other words, we only select from the atomic operations of the 15 properties. The reason is we notice that the formatting requirements associated with the four properties seldom co-occur with the other attributes. (2) At each combing, there will not be multiple operations with the same properties. (3) The range of N is from 2 to 12. For instance, when N equals 10, the target formatting instruction will involve atomic operations on 10 properties.

To avoid assessment bias due to similar test samples, inspired by recent work (Liu et al., 2023b), we eliminate combinations that are more similar to

[2] We take Word API 1.1, which is developer-oriented and supports most versions of Microsoft Word.

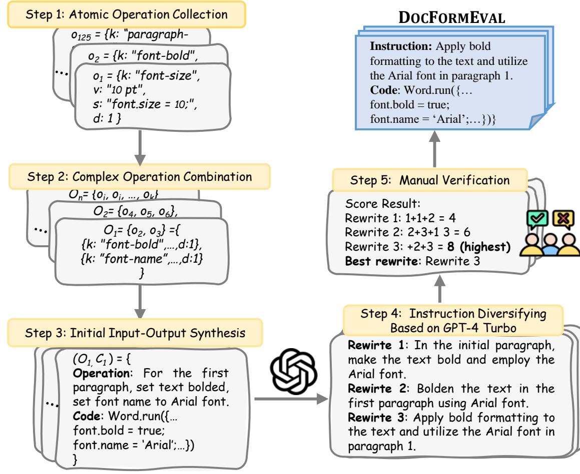

Figure 2: Overview of DocFormEval construction pipeline.

others. Specifically, given that each atomic operation combination is a set, we utilize the Jaccard similarity measure to measure the similarity between combinations. Thus, the similarity between any two combinations O_i and O_j is computed as follows:

$$\text{Similar}(O_i, O_j) = \frac{|O_i \cap O_j|}{|O_i \cup O_j|} \leq 0.5 \tag{2}$$

By performing the actions mentioned above, we can obtain the initial set of document formatting backbones $\mathcal{O}^0 = \{O_1, O_2, ..., O_M\}$, where O_i may be an atomic operation or a collection of atomic operations.

Initial Input-Output Synthesis We can only automatically execute a formatting operation when provided with the position of the object to be formatted. The position is either given by the user through mouse selection or expressed in the formatting instruction, e.g., "set the font size of the third paragraph to 5". To simulate the position acquisition during document formatting, for each atomic operation in each backbone O_i in $\mathcal{O}^0$, we randomly assign a paragraph position $p \in [0, 10]$. If p equals 0, it means that the user selects the position of the operation via the mouse, and thus, it does not exist in the formatting instruction. In the above way, we obtain the backbone set $\mathcal{O}$ of document formatting instructions.

By splicing the code snippets of the formatting operations of each backbone O_i in $\mathcal{O}$, we obtain its corresponding formatting code snippet C_i. Furthermore, we convert each backbone O_i into formatting instruction I_i^t by a template-based heuristic method. Finally, we obtain the initial evaluation dataset $\mathcal{D}^0 = \{(I_i^t, C_i) | I_i^t \in \mathcal{I}^t, C_i \in \mathcal{C}\}$, where $\mathcal{C}$ is the set of formatting codes. The example with a not executable C is discarded. Please refer to Appendix A.1 for more details.

Instruction Diversifying Based on GPT-4 Turbo We note that the formatting instructions in $\mathcal{I}^t$ constructed entirely based on templates are too uniform in wording, with similar representations among different instructions. Inspired by recent works, this paper rewrites template-based instructions using the powerful GPT-4 Turbo to enhance their diversity. Since LLMs tend to get stuck in local optimal solutions (Wang et al., 2022c), rewriting formatting instructions only once may lead to semantic errors, so we rewrite each template instruction 3 times. For more details on instruction rewriting, please refer to Appendix A.2.

Manual Verification To improve the quality of the evaluation dataset, we manually score and calibrate the rewritten formatting instructions. Specifically, three annotators are assigned to score each rewritten instruction for accuracy and appropriateness. A score of 3 is the highest, while 0 is given if the rewriting does not match the original semantics. The rewritten formatting instruction with the highest average score from the three annotators is then selected as the final instruction for that example.

From the above data construction pipeline, we develop the evaluation dataset DocFormEval. It contains 1,911 high-quality document formatting samples, each consisting of a natural language formatting instruction I and a Javascript code snippet C that can run in a Microsoft Word environment $\mathcal{E}$ to accomplish I. Due to page limitations, we provide a detailed statistical analysis of DocFormEval in Appendix A.3. Please refer to it for more details.

4 Text-to-Format

Previous researches (Zan et al., 2022; Wu et al., 2023; Béchard and Ayala, 2024) demonstrate that the retrieval-augmented generation methods (RAG) can help LLMs reduce hallucinations and generate more accurate code for specific tasks. Inspired by these efforts, we introduce a document formatting code generation approach called Text-to-Format. This approach is driven by LLMs and combines multiple prompting techniques. As shown in Figure 3, the Text-to-Format con-

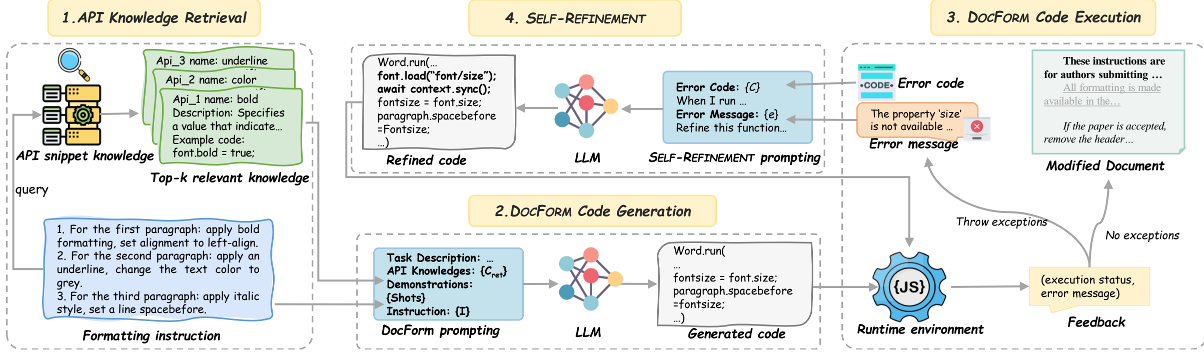

Figure 3: The architecture of our proposed TEXT-TO-FORMAT.

sists of four stages. First, it utilizes vector retrieval techniques to retrieve API knowledge semantically associated with the current formatting instruction from a pre-built knowledge base. Second, it generates formatting code snippets conforming to the syntax rules required by Microsoft Word based on the instruction and the retrieved knowledge. Then, it executes the code in our constructed runtime environment for formatting. Lastly, if an exception is thrown in code execution, the error code is repaired using the self-refinement mechanism. We will describe these four parts in detail as follows.

4.1 API Knowledge Retrieval

LLMs may have yet to see API knowledge related to DOCFORM during their training. Therefore, directly prompting LLMs may produce the output C containing hallucinations. Inspired by RAG works, we retrieve API knowledge related to user formatting instructions to reduce this issue. In fact, the Microsoft Word JavaScript APIs are only on the Office website and cannot be accessed offline, so we first manually built an API snippet knowledge base $C_{api} = \{c^1_{api}, c^2_{api}, \dots, c^n_{api}\}$ on it. Each snippet c^k_{api} contains relevant information about a Word formatting operation API, including its usage description and example code. Secondly, they are embedded by a retrieval model R, saved in the popular vector database Faiss (Johnson et al., 2019). Lastly, we input formatting instruction I as a query, retrieving the top k relevant knowledge C_{ret} from C_{api}. This process yields a set of retrieved knowledge snippets

$$C_{ret} = R(I, k, C_{api}) \tag{3}$$

4.2 DOCFORM Code Generation

We elaborate the prompts to elicit LLMs better to generate formatting code that meets user formatting requirements as well as programming specifications. The prompt $P = T(C_{ret}, S, I)$ is a triple, where C_{ret} is the set of retrieved API knowledge, S is the demonstrations when enabling few-shot leaning, and I is the formatting instruction from the user. Lastly, we utilize P to guide LLMs generating corresponding code C. The structure of the document formatting code generation prompt P is shown below:

```
Task Description: Develop a Node.js plugin for
Microsoft Word that dynamically generates...
APIs Knowledge:  Api knowledges you may need,
each api structure may include ...
Api Knowledge list: {C_ret}
Input-Output Sample: {S}
Input: {I}
Output:
```

4.3 DOCFORM Code Execution

First, we build a runtime environment $\mathscr{E}$ based on the Microsoft Office Add-ins. This environment can be used to execute the generated code to automatically modify the document format, which helps bridge the gap between the code and the formatted document. It can provide two types of feedback: execution status $F = \{\texttt{true}, \texttt{false}\}$ and error message e. If F is true, it indicates that the formatting code is executed successfully. Otherwise, it means that there is an exception.

4.4 SELF-REFINEMENT Mechanism

To repair execution exceptions in generated code, we introduce a SELF-REFINEMENT mechanism building upon prior work (Jiang et al., 2023a; Madaan et al., 2024). We collect the error code C^e with error message e and input them along the LLMs for SELF-REFINEMENT. By running the repaired code in the runtime environment, we can get an execution status F' and an error message e'. If F' is still $false$, repeat SELF-REFINEMENT until

F' turns to *true* or the maximum iteration times is reached. The structure of SELF-REFINEMENT prompt is shown below:

```
Wrong Code: {C^e}
When I run this function, I meet the following
error,error Message: {e}
Help me refine the code.
You should only output the codes without any
explanation and natural language.Wrap your code
with '''.
```

5 Experimental Setup

5.1 Baselines

In addition to directly prompting LLMs to generate document formatting code (PROMPT), we plan to explore the performance of the following three popular prompting engineering methods in code generation.

- SELF-DEBUGGING (Chen et al., 2023) can learn from debugging demonstrations within prompting, reducing the errors in generated code. They propose three debugging feedback methods, including "Simple," "Unit Testing," and "Explanation." In this paper, the generated code is not suitable for unit testing. Therefore, we use "Explanation" as feedback.
- DOC PROMPTING (Wu et al., 2023) teaches models to generated specific code by reading API knowledge. In this paper, we use DOC PROMPTING to reduce the hallucinations in generated code by retrieving API knowledge.
- SELF-REFINEMENT (Jiang et al., 2023a) allows LLMs to repair the generated code when it throws exceptions. In this paper, we use this mechanism to improve the fault tolerance of LLMs.

5.2 Evaluation Metrics

Through observation, we find two types of format code errors. The first is an execution exception, meaning the code contains syntax errors and does not meet the specification. An exception occurs when it is executed in the runtime environment, causing the process to terminate prematurely. The other is a formatting error, suggesting the code can be executed smoothly, but the document formatting results do not match the user's requirements. To sum up, to better understand the shortcomings of the method, we use the following three evaluation indicators to evaluate the TEXT-TO-FORMAT performance on the DOCFORMEVAL: (1) Formatting Accuracy (FA): it is the proportion of generated code that executes successfully and meets requirements. (2) Execution Exception Rate (EER): it denotes the proportion of generated code that executes successfully but does not meet requirements.(3) Formatting Error Rate (FER): it is the proportion of generated code that throws an exception during execution. Particularly, their binding relationship is $FA + EER + FER = 100\%$. Please refer to Appendix A.4 for detailed calculations.

MODEL	Version	Context Length	Param
Locally Deployed			
Llama3(AI@Meta, 2024)	Instruct	8k	8B
Qwen2(Bai et al., 2023a)	Instruct	128k	7B
CodeQwen1.5(Bai et al., 2023b)	Chat	32k	7B
API-Invoked			
DeepSeek-Coder (DeepSeek-AI, 2022)	-	32K	33B
DeepSeek-Chat (DeepSeek-AI, 2024)	V2	32K	236B
Gemini Pro (Team et al., 2023)	1.0	32K	-
GPT-3.5 Turbo (OpenAI, 2022)	0125	16k	-
GPT-4 Turbo (Achiam et al., 2023)	2024-04-09	128k	-

Table 1: Statistics on the LLMs utilized in the experiment.

5.3 Implementation Details

We examine the performance of eight large models in generating DOCFORM code, which can be divided into local deployment and API invocation according to their invocation ways. The details are displayed in Table 1. DOC PROMPTING is a retrieval-augmented generation method, and the text vectorization model dramatically influences its performance. Therefore, we utilize Faiss (Johnson et al., 2019) as a vector storage database and test the effectiveness of four top-ranked semantic vectorization models on the validation set. The four models are the E5 (Wang et al., 2022a), BEG (Xiao et al., 2023), GTE (Li et al., 2023), and MABAI (Lee et al., 2024). Finally, we choose E5, which performs best on the validation set.

Due to the limited page number, we put the hyperparameter settings for different large models, prompting strategies, the shot number of few-shot learning, and the search range for these parameters in Appendix A.5.

6 Experimental Results

6.1 Main Result

Table 2 summarizes the performance of different TEXT-TO-FORMAT methods on the DOCFORMEVAL test set based on different prompting

Methods	Local Deployed			API-invoked				
	LLama3 8B	Qwen2 7B	CodeQwen1.5 7B	DeepSeek-Coder 33B	DeepSeek-Chat 236B	Gemini Pro -	GPT-3.5 Turbo -	GPT-4 Turbo -
Prompt	4.07	0.00	11.24	7.71	5.61	2.74	13.28	**34.41**
Self-Debugging	30.12	29.49	38.86	30.43	41.68	42.64	27.94	**64.30**
Doc Prompting	62.12	38.14	36.87	56.06	69.78	59.40	39.61	**81.26**
Prompting$_{few\text{-}shot}$	45.93	64.41	63.58	79.38	**83.57**	75.67	63.44	81.46
Doc Prompting$_{few\text{-}shot}$	3.58	65.35	61.67	77.04	**89.81**	82.49	75.52	88.34
With Self-Refinement								
Prompt	4.14$_{\uparrow 0.07}$	2.48$_{\uparrow 2.48}$	11.77$_{\uparrow 0.53}$	10.25$_{\uparrow 2.54}$	7.21$_{\uparrow 1.60}$	5.47$_{\uparrow 0.73}$	22.08$_{\uparrow 8.80}$	**51.47**$_{\uparrow 17.1}$
Self-Debugging	44.85$_{\uparrow 14.7}$	33.15$_{\uparrow 3.66}$	40.55$_{\uparrow 1.69}$	36.23$_{\uparrow 5.80}$	49.82$_{\uparrow 8.14}$	51.32$_{\uparrow 8.68}$	37.55$_{\uparrow 9.61}$	**77.72**$_{\uparrow 13.4}$
Doc Prompting	62.20$_{\uparrow 0.08}$	40.19$_{\uparrow 2.05}$	37.03$_{\uparrow 0.16}$	62.21$_{\uparrow 6.15}$	75.74$_{\uparrow 5.96}$	68.96$_{\uparrow 9.56}$	44.92$_{\uparrow 5.31}$	**85.47**$_{\uparrow 4.21}$
Prompting$_{few\text{-}shot}$	52.20$_{\uparrow 6.27}$	66.23$_{\uparrow 1.82}$	64.71$_{\uparrow 1.13}$	82.39$_{\uparrow 3.01}$	86.66$_{\uparrow 3.09}$	82.04$_{\uparrow 6.37}$	74.62$_{\uparrow 11.2}$	**89.20**$_{\uparrow 7.74}$
Doc Prompting$_{few\text{-}shot}$	4.31$_{\uparrow 0.73}$	68.13$_{\uparrow 2.78}$	61.83$_{\uparrow 0.16}$	81.53$_{\uparrow 4.19}$	**91.54**$_{\uparrow 1.73}$	83.09$_{\uparrow 0.60}$	82.72$_{\uparrow 7.20}$	91.43$_{\uparrow 3.09}$

Table 2: Formatting Accuracy(%) of Text-to-Format driven by various prompting strategies and large language models. ↑ denotes the improvement in accuracy after introducing the self-refinement mechanism.

strategies and LLMs. In particular, Doc Prompting$_{few\text{-}shot}$ + Self-Refinement performs best. When driven by GPT-4 Turbo or DeepSeek-Chat, it can achieve a formatting accuracy of over 91%. In the rest of this section, we explore how to obtain a superior Text-to-Format method in an offline or online environment by comparing and analyzing in the following two aspects.

Effect of Different Prompting Strategies We first observe that zero-shot performance is poor when LLMs are directly prompted to generate document formatting codes (Prompt), regardless of which LLM is driving. Specifically, most models are less than 10% accurate, and some smaller models, such as Qwen2-7B, do not work. Even the powerful GPT-4 turbo only achieves 34.41% accuracy. We argue that this is mainly due to their lack of knowledge about document formatting code rather than their code-generation capabilities.

Then, as more sophisticated prompting techniques are applied, the performance of Text-to-Format rises. Taking the example of the Text-to-Format driven by the open-source model Qwen2, it exceeds Prompt by +26.05% and +58.05%, respectively, after the introduction of Self-Debugging and Doc Prompting strategies, After further introducing the Self-Refinement mechanism, the performance is further improved by +1.69% and +0.16% on the previous basis. When Text-to-Format is driven by commercial models, these strategies are also effective, where the zero-shot performance of Text-to-Format based on GPT-4 Turbo improves from 34.41% at most to 81.26%. Once again, these classic prompting strategies for code-generation strategies are proven to be effective.

We argue that these prompting strategies are effective partly because they all introduce knowledge related to document formatting to varying degrees, which may also account for the different improvements they bring. The explanation is as follows. The Self-Debugging method introduces a few formatting code examples in its "Explanation." Doc Prompting introduces more information about the APIs needed to generate formatting code. Self-Refinement builds on the knowledge introduced in the first two methods by introducing more knowledge about negative examples when generating code that cannot be run directly.

Effect of Few-shot Learning As we can see, few-shot learning benefits for the three strategies, with the most significant improvement for the Prompt method. However, there are a few noteworthy points. First, we notice that the performance of LLama3-based Text-to-Format decreases significantly. Second, the relationship between the advantages and disadvantages of the prompting strategies is reversed in some models. For example, when the base model is DeepSeek-Coder, the Doc Prompting$_{few\text{-}shot}$ strategy without the Self-Refinement mechanism is inferior to Prompting$_{few\text{-}shot}$, dropping by −2.34%. We believe that the reason for the above less intuitive results may be that few-shot learning is also a means of introducing knowledge related to the formatting task, as well as making the input context longer (17 shots for all methods). It may lead to the fact that, after combining the internal knowledge of the model, the knowledge brought in by the prompting strategy, and the shots, the Text-to-Format bottleneck may be not only the lack of knowledge about the formatting task but also the ability to deal with longer contexts.

6.2 Further Analysis

We analyze the impact of formatting complexity in two aspects. One aspect is based on the number of properties contained in the input instruction, while the other is based on the instruction complexity, which is the minimal number of lines of formatting code that need output.

As shown in Figure 4, the execution accuracy of all models decreases as the number of properties in the instructions increases. However, the rate and reason for the decrease are less consistent. Among them, Deepseek-Chat and GPT-4 Turbo, the two best-performing models, show a slower rate of decrease in accuracy, and the exception rate almost remains stable. This implies that as the input is improved, its ability to follow instructions does not decline, and it can stably output code that conforms to the specified syntax rules. It is evident that the primary cause of the decrease in accuracy is the higher wrong rate resulting from more challenging formatting operations. Similarly, we find that the decrease in execution accuracy of Gemini Pro is mainly due to the elevated exception rate, indicating that the quality of the generated code decreases when it processes more formatting operations at once. Differently, the decrease in execution accuracy of GPT-3.5 Turbo is caused by a combination of elevated exception and error rates. Overall, analyzing the exception and wrong rates of TEXT-TO-FORMAT helps us deeply understand the method's shortcomings and guides us on optimizing it.

Furthermore, Figure 5 shows that the formatting accuracy of the model decreases as the complexity increases. However, the rate of performance degradation is much lower for GPT-4 and DeepSeek than for Gemini Pro and GPT-3.5 Tubro, suggesting that the stronger the LLM, the more stable it is.

In addition, we also explore the impact of retrieval models, the number of demonstrations used in few-shot learning, and the impact of different shots. Please refer to Appendices A.6 ~ A.8 for detailed information.

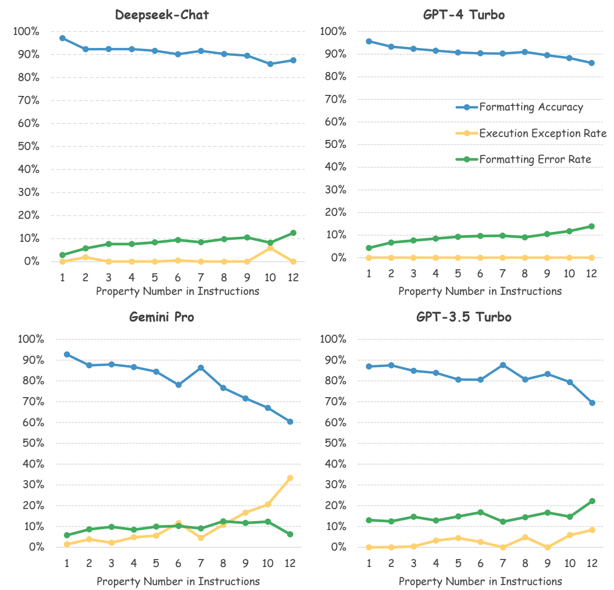

Figure 4: Impact of the number of properties involved in the formatting instruction on performance.

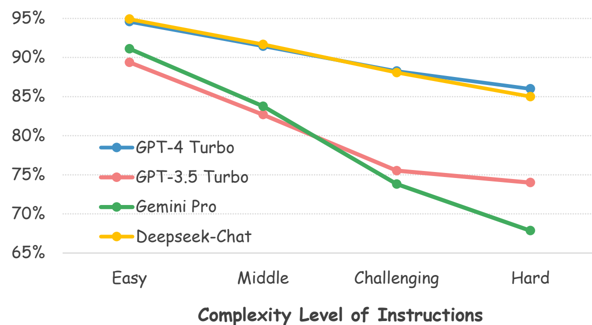

Figure 5: Effect of instruction complexity on formatting accuracy. The complexity is defined as the minimal number of lines of formatting code that need output.

7 Conclusion

This paper highlights the document formatting task (DOCFORM) in Microsoft Word, a valuable application scenario for code generation. For DOCFORM, we first propose TEXT-TO-FORMAT, an automatic document formatting algorithm driven by various prompting strategies. TEXT-TO-FORMAT takes a textual formatting instruction and then generates a code that can be run in Microsoft Word to format the content in a document. Meanwhile, we build an evaluation specification including a high-quality evaluation dataset DOCFORMEVAL, a code runtime environment, and valuation metrics to evaluate automatic document formatting approaches and advance the document formatting task. Extensive experimental results on DOCFORMEVAL demonstrate that current open-source and commercial models contain little knowledge about DOCFORM. In addition,the performance of the prompt strategy is related to the amount of knowledge it introduces related to DOCFORM. However, in a few-shot setting, introducing too many samples may pose challenges to the underlying LLM's ability to handle long contexts, even if the input context length is within its acceptable range.

Limitations

The limitations of our work are summarized in the following aspects.

First, the proposed dataset only provides an evaluation dataset and no training data. Nonetheless, we consider the training data may not be necessary because the Text-to-Form method based on open-source Qwen2 (7B) and DeepSeek-Chat (236B) achieves 68.13% and 91.54% accuracies after introducing various prompting and the retrieval-argument strategies.

In addition, we do not explore larger open-source LLMs (e.g., LLaMA3-70B, Qwen2-72B) due to computational resource constraints. Nonetheless, we believe that the performance of TEXT-TO-FORMAT based on Deep-seek Chat (236B) is instructive, as Deep-seek Chat is one of the best open-source models currently available.

We conduct a detailed analysis of the token consumption of various prompting strategies on GPT-4 Turbo. The results are displayed in Figure 6. As we can see, although the DOC PROMPTING$_{few\text{-}shot}$ method can achieve 91% formatting accuracy in GPT-4 Turbo and DeepSeek-Chat, the token consumption of this approach is also remarkable. Moreover, the average input tokens for the DOC PROMPTING$_{few\text{-}shot}$ method exceed 3700 tokens on GPT-4 Turbo, which is a significant cost. As shown in Figure 7, high token consumption is caused by shot demonstrations and API documentation in the input instruction. This high consumption may be due to the LLM's lack of knowledge in document formatting. We will consider exploring algorithms that will consume fewer tokens in the future.

Finally, to support offline deployment, this paper refers to fundamental Microsoft development documentation (Word API 1.1). The text-to-form method only formats the text content in the document, such as paragraph and font formats. Other types of data content in the document, such as images, tables, and layout, are not supported.

Ethics Statement

This paper proposes an automatic formatting method, TEXT-TO-FORMAT, and an evaluation dataset, DOCFORMEVAL, for the document formatting task. On the one hand, the document formatting task only modifies the format of the content in the document and does not modify the content itself. On the other hand, Both TEXT-TO-FORMAT and DOCFORMEVAL refer to Microsoft's publicly available Word development documentation (Word API 1.1). Therefore, there are no ethical violations in either the methodology or the data presented in this paper.

During the construction of DOCFORMEVAL, we used GPT-4 Turbo to enrich the diversity of document formatting instructions obtained by synthesis. Specifically, it costs 5733 API calls to GPT-4 Turbo. At the manual verification stage, we engage five graduate-level Microsoft Word developers from our team as annotators. Among them, three annotators evaluate rewritten results, while two review annotated data. Lastly, we request all annotators to filter out the instructions generated by GPT-4 Turbo that leak personal privacy, contain social bias, or contain harmful content.

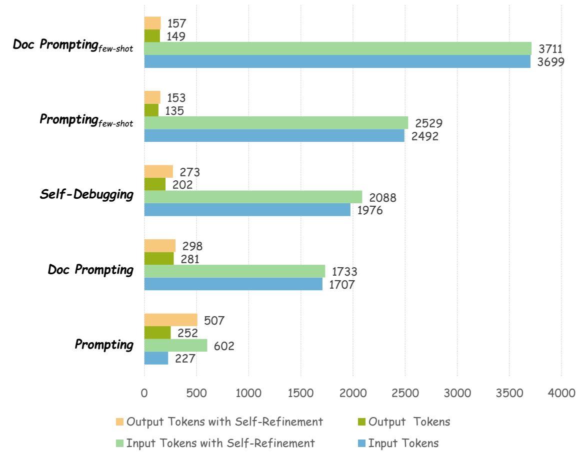

Figure 6: Token consumption for various prompting strategies on GPT-4 Turbo.

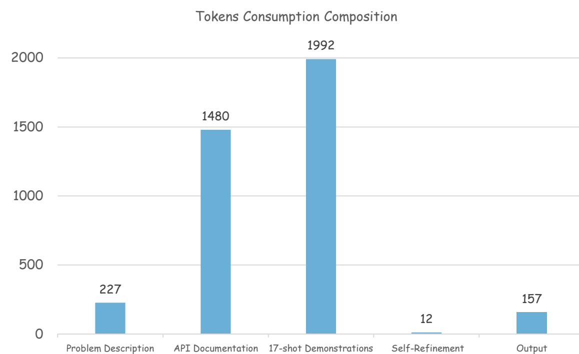

Figure 7: Token consumption of different parts for DOC PROMPTING$_{few\text{-}shot}$ strategy on GPT-4 Turbo.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

AI@Meta. 2024. Llama 3 model card.

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, Keming Lu, Jianxin Ma, Rui Men, Xingzhang Ren, Xuancheng Ren, Chuanqi Tan, Sinan Tan, Jianhong Tu, Peng Wang, Shijie Wang, Wei Wang, Shengguang Wu, Benfeng Xu, Jin Xu, An Yang, Hao Yang, Jian Yang, Shusheng Yang, Yang Yao, Bowen Yu, Hongyi Yuan, Zheng Yuan, Jianwei Zhang, Xingxuan Zhang, Yichang Zhang, Zhenru Zhang, Chang Zhou, Jingren Zhou, Xiaohuan Zhou, and Tianhang Zhu. 2023a. Qwen technical report. *arXiv preprint arXiv:2309.16609*.

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, Keming Lu, Jianxin Ma, Rui Men, Xingzhang Ren, Xuancheng Ren, Chuanqi Tan, Sinan Tan, Jianhong Tu, Peng Wang, Shijie Wang, Wei Wang, Shengguang Wu, Benfeng Xu, Jin Xu, An Yang, Hao Yang, Jian Yang, Shusheng Yang, Yang Yao, Bowen Yu, Hongyi Yuan, Zheng Yuan, Jianwei Zhang, Xingxuan Zhang, Yichang Zhang, Zhenru Zhang, Chang Zhou, Jingren Zhou, Xiaohuan Zhou, and Tianhang Zhu. 2023b. Qwen technical report. *arXiv preprint arXiv:2309.16609*.

Patrice Béchard and Orlando Marquez Ayala. 2024. Reducing hallucination in structured outputs via retrieval-augmented generation. *arXiv preprint arXiv:2404.08189*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, and et al. 2021. Evaluating large language models trained on code.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*.

Tristan Coignion, Clément Quinton, and Romain Rouvoy. 2024. A Performance Study of LLM-Generated Code on Leetcode. In *EASE'24 - 28th International Conference on Evaluation and Assessment in Software Engineering*, Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering (EASE'24), Salerno, Italy.

Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C Desmarais, and Zhen Ming Jack Jiang. 2023. Github copilot ai pair programmer: Asset or liability? *Journal of Systems and Software*, 203:111734.

DeepSeek-AI. 2022. Deepseek coder - an ai programming assistant. Accessed: 2022-03-01.

DeepSeek-AI. 2024. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model.

Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2023. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation. *arXiv preprint arXiv:2308.01861*.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*.

Shuyang Jiang, Yuhao Wang, and Yu Wang. 2023a. Selfevolve: A code evolution framework via large language models. *arXiv preprint arXiv:2306.02907*.

Zhengbao Jiang, Frank F Xu, Luyu Gao, Zhiqing Sun, Qian Liu, Jane Dwivedi-Yu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023b. Active retrieval augmented generation. *arXiv preprint arXiv:2305.06983*.

Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 7(3):535–547.

X. Konglong. 2023. Gw project. `https://github.com/xkonglong`. Accessed: 2024-05-24.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Scott Wen tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2022. Ds-1000: A natural and reliable benchmark for data science code generation. *ArXiv*, abs/2211.11501.

Sean Lee, Aamir Shakir, Darius Koenig, and Julius Lipp. 2024. Open source strikes bread - new fluffy embeddings model.

Zehan Li, Xin Zhang, Yanzhao Zhang, Dingkun Long, Pengjun Xie, and Meishan Zhang. 2023. Towards general text embeddings with multi-stage contrastive learning. *arXiv preprint arXiv:2308.03281*.

Chao Liu, Xuanlin Bao, Hongyu Zhang, Neng Zhang, Haibo Hu, Xiaohong Zhang, and Meng Yan. 2023a. Improving chatgpt prompt for code generation. *arXiv preprint arXiv:2305.08360*.

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. *Advances in Neural Information Processing Systems*, 36.

Wei Liu, Weihao Zeng, Keqing He, Yong Jiang, and Junxian He. 2023b. What makes good data for alignment? a comprehensive study of automatic data selection in instruction tuning. *arXiv preprint arXiv:2312.15685*.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. 2024. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36.

Aman Madaan, Shuyan Zhou, Uri Alon, Yiming Yang, and Graham Neubig. 2022. Language models of code are few-shot commonsense learners. *arXiv preprint arXiv:2210.07128*.

Yu Nong, Mohammed Aldeen, Long Cheng, Hongxin Hu, Feng Chen, and Haipeng Cai. 2024. Chain-of-thought prompting of large language models for discovering and fixing software vulnerabilities. *arXiv preprint arXiv:2402.17230*.

OpenAI. 2022. Chatgpt.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Weijia Shi, Sewon Min, Michihiro Yasunaga, Minjoon Seo, Rich James, Mike Lewis, Luke Zettlemoyer, and Wen-tau Yih. 2023. Replug: Retrieval-augmented black-box language models. *arXiv preprint arXiv:2301.12652*.

Hongjin Su, Shuyang Jiang, Yuhang Lai, Haoyuan Wu, Boao Shi, Che Liu, Qian Liu, and Tao Yu. 2024. Arks: Active retrieval in knowledge soup for code generation. *arXiv preprint arXiv:2402.12317*.

Ningzhi Tang, Meng Chen, Zheng Ning, Aakash Bansal, Yu Huang, Collin McMillan, and Toby Jia-Jun Li. 2024. A study on developer behaviors for validating and repairing llm-generated code using eye tracking and ide actions. *arXiv preprint arXiv:2405.16081*.

Gemini Team, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, et al. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*.

OpenDevin Team. 2024. Opendevin: An open platform for ai software developers as generalist agents. `https://github.com/OpenDevin/OpenDevin`. Accessed: ENTER THE DATE YOU ACCESSED THE PROJECT.

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

Xingyao Wang, Sha Li, and Heng Ji. 2022b. Code4struct: Code generation for few-shot event structure prediction. *arXiv preprint arXiv:2210.12810*.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2022c. Self-instruct: Aligning language models with self-generated instructions. *arXiv preprint arXiv:2212.10560*.

Sijin Wu, Dan Zhang, Teng Hu, and Shikun Feng. 2023. Docprompt: Large-scale continue pretrain for zero-shot and few-shot document question answering. *arXiv preprint arXiv:2308.10959*.

Shitao Xiao, Zheng Liu, Peitian Zhang, and Niklas Muennighoff. 2023. C-pack: Packaged resources to advance general chinese embedding.

Fangyuan Xu, Weijia Shi, and Eunsol Choi. 2023. Recomp: Improving retrieval-augmented lms with compression and selective augmentation. *arXiv preprint arXiv:2310.04408*.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. *arXiv preprint arXiv:1809.08887*.

Yue Yu, Yuchen Zhuang, Jieyu Zhang, Yu Meng, Alexander J Ratner, Ranjay Krishna, Jiaming Shen, and Chao Zhang. 2024. Large language model as attributed training data generator: A tale of diversity and bias. *Advances in Neural Information Processing Systems*, 36.

Daoguang Zan, Bei Chen, Zeqi Lin, Bei Guan, Yongji Wang, and Jian-Guang Lou. 2022. When language model meets private library. *arXiv preprint arXiv:2210.17236*.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023a. Repocoder: Repository-level code completion through iterative retrieval and generation. *arXiv preprint arXiv:2303.12570*.

Ziyin Zhang, Chaoyu Chen, Bingchang Liu, Cong Liao, Zi Gong, Hang Yu, Jianguo Li, and Rui Wang. 2023b. A survey on language models for code. *arXiv preprint arXiv:2311.07989*.

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 evaluations on humaneval-x. *arXiv preprint arXiv:2303.17568*.

Terry Yue Zhuo. 2023. Large language models are state-of-the-art evaluators of code generation. *arXiv preprint arXiv:2304.14317*.

A Appendix

A.1 DocFormEval Construction Details

Properties Selection To simulate real user scenarios in DOCFORMEVAL, we consider two key aspects in selecting properties related to paragraph and font formatting. Firstly, we select properties of paragraph and font based on the formatting functions already provided in Microsoft Word. Secondly, we further check whether these functions could be supported through Microsoft's Office Add-ins. We can generate code that calls these APIs to use the formatting adjustment feature when the Add-ins provide the corresponding API. Only properties that can be manipulated by code are retained. By the above steps, we obtain 19 available properties, which basically cover most formatting needs.

Actually, the number is limited because many formatting operations are provided within the software. However, Microsoft does not expose the API for these operations or allow them to be read but not written. This restriction limits the number of properties we can choose.

Synthesis Template For $O_i \in \mathscr{O}^0$, we provide a specific example to demonstrate the synthesis template. For instance, $O = \{o_1, o_2, o_3\}$, it is assigned the second paragraph, $p = 2$. Detailed O is expressed as follows:

```
O = {
o1 = {
k1: "font-size",
v1: "10 pt",
s1: "font.size = 10;",
d1: 1 },
o2 = {
k2: "paragraph-spaceBefore",
v2: "1 line",
s2: """
font.load("size");
await context.sync();
paragraph.spaceBefore = 3*font.size;
""",
d2: 3},
o3 = {
k3: "font-color",
v3: "red",
s3: "font.color = "red";,
d3: 1 },
}
```

The instruction synthesis template is as follows:

```
For paragraph {p}, please set the {k1} to {v1},
please set the {k2} to {v2},
...
please set the {kn} color to {vn}.
```

Then we fill the template with the corresponding values from O, obtaining the instruction I as follows:

```
For paragraph 2, please set the font's size to
10 pt, please set the paragraph's spaceBefore
to 1 line, please set the font's color to red.
```

Similarly, the code synthesis template is followed:

```
Word.run(async function (context) {
const paragraphs=context.document.body.paragraphs;
paragraphs.load("$none");
await context.sync();
const paragraph = paragraphs.items[{p-1}];
const font = paragraph.font;
{c1}
{c2}
...
{cn}
await context.sync();
});
```

Then we fill the code template with the corresponding values from O, obtaining the code C as follows:

```
Word.run(async function (context) {
const paragraphs=context.document.body.paragraphs;
paragraphs.load("$none");
await context.sync();
const paragraph = paragraphs.items[1];
var font = paragraph.font;
font.size = 10;
font.load("size");
await context.sync();
paragraph.spaceBefore = 3*font.size;
"font.color = "red"
await context.sync();
});
```

Instruction Complexity We use the number of lines of code needed to complete the instructions to assess the complexity of the instructions indirectly. Intuitively, the more complex the instructions, the more formatting operations they involve, which means more lines of code are needed to complete these operations. For instance, the formatting instruction I is converted by backbone O. Thus, the complexity of instruction I is the number of lines of key code in C, which is the sum of all complexity d_i in backbone O.

Position Assignment When randomly assigning positions, we should consider the varying frequencies and control the weights of the random assignments, as shown in Table 3. The position of the paragraph is ten times the mouse-selected position. The weight of Paragraph 1 is set to 1.5, while others are set to 1 because the first paragraph is usually a title, more likely to be modified.

P	Position	Weight	Number
0	`Mouse-Selected`	1.5	355
1	`Paragraph 1`	1.5	309
2	`Paragraph 2`	1	186
3	`Paragraph 3`	1	164
4	`Paragraph 4`	1	179
5	`Paragraph 5`	1	197
6	`Paragraph 6`	1	188
7	`Paragraph 7`	1	190
8	`Paragraph 8`	1	198
9	`Paragraph 9`	1	158
10	`Paragraph 10`	1	187

Table 3: Property distribution of DocFormEval.

A.2 Diversifying Prompting

For instructions synthesized based on rule templates, we rewrite them three times by GPT-4 Turbo. The prompting of rewriting is as follows:

```
Rewrite the Word document formatting
instructions provided in original instruction
to ensure the language is natural, fluid, and
varied. The original intent and instructions
must be preserved.
Original instruction: {Instruction}
Output the revised instruction below:
"""
```

Each initial instruction will be rewritten three times by GPT-4 Turbo. Each rewritten instruction will be scored by three annotators independently, with scores ranging from 0 to 3. The detailed description of the $0 \sim 3$ scoring is as follows:

```
0: The rewritten result does not have the
same meaning as the original instruction,
expressed incorrectly. Or it contains biased
and discriminatory output.
1: The rewritten result has the same meaning
as the original instruction but is overly
verbose.
2: The rewritten result has the same meaning as
the original instruction and generally aligns
with human user habits.
3: The rewritten result has the same meaning as
the original instruction, is clearly expressed,
and perfectly aligns with human user habits.
```

A.3 Statistics of DocFormEval

To establish a more comprehensive understanding of DocFormEval, we investigate it from four aspects: the property distribution of formatting operations, the formatting instruction length distribution, the property number distribution per instruction, and instruction complexity.

First, as shown in Table 4, the percentage of properties at the word and para-

Property	Number	Proportion(%)
Paragraph Property		
`alignment`	608	6.55
`leftIndent`	456	4.91
`firstLineIndent`	646	6.96
`lineSpacing`	665	7.16
`outlineLevel`	684	7.37
`rightIndent`	613	6.60
`spaceBefore`	635	6.69
`spaceAfter`	621	6.84
Font Property		
`bold`	603	6.49
`italic`	611	6.58
`color`	645	6.95
`underline`	610	6.57
`name`	611	6.58
`size`	646	6.96
`highlightColor`	614	6.61
`strikeThrough`	4	0.04
`doubleStrikeThrough`	4	0.04
`subscript`	5	0.05
`superscript`	5	0.05

Table 4: Property distribution of DocFormEval.

graph level is similar, showing an even distribution. The low percentage of properties `strikeThrough`, `doubleStrikeThrough`, `subscript`, and `superscript` is because they are not combined with other atomic formatting operations in the combination stage, considering that they are generally used separately.

Second, looking at the length distribution of the formatting instructions in Figure 8, we find that most instruction lengths are 30-70 words, with fewer extremely short or long inputs. Overall, the instruction length distribution approximates a normal distribution.

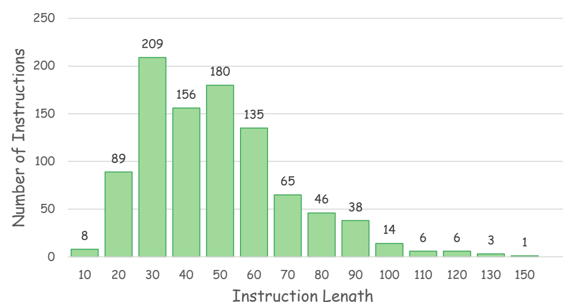

Figure 8: Length distribution of instructions in DocFormEval.

Third, the property number distribution per instruction in Figure 9. The number of properties involved in instruction is concentrated between 3 and 6. The number of instructions with property counts between 4 and 7 is relatively small. Because

Figure 9: Property number distribution of instructions in DOCFORMEVAL.

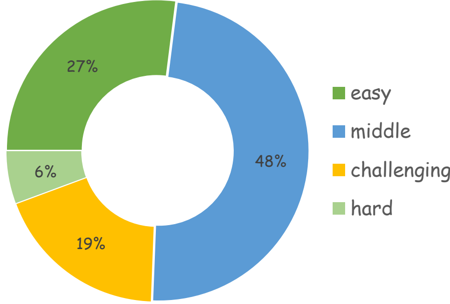

Figure 10: Complexity distribution of instructions in DOCFORMEVAL.

we filtered them out when controlling for similarity between instructions using the Jaccard similarity.

Lastly, the instruction complexity distribution is shown in Figure 10. Complexity defines the essential lines of code required to implement this instruction. Instructions requiring 1-5 lines of code are considered easy-level. Those with 6-10 lines are middle-level, while instructions requiring 11-15 lines are challenging-level. Instructions with more than 15 lines are hard-level.

A.4 Metric Formulation

We calculate the Formatting Accuracy (FA) by comparing the properties of the text in the Word documents after executing both the ground-truth formatting code and the generated code. Specifically, each example in the DOCFORMEVAL dataset includes a natural language formatting instruction and a ground-truth formatting code. For the text in Word, we can extract its style property set using the Microsoft Word API. The property set is denoted as $Style = \{p^i = v^i\}$, where pi represents a text property, and vi represents its value. For example, font-size=5 indicates that the font size of the text is 5. By comparing the property set $Style^{ground-truth}$ after executing the ground-truth formatting code with the property set $Style^{generated}$ after executing the generated code, we can easily determine whether the formatting results of the generated code meet the instructions. The detailed formulation is as follows:

We firstly represent DOCFORMEVAL as $D = D_1, D_2, ..., D_n$. For $\forall d_i \in d$, $d_i = (I_i, O_i, C_i)$, and I_i denotes the instruction. $O_i = \{o_{i1}, o_{i2}, o_{om}\}$, it denotes the backbone of instruction I_i. C_i is the executable formatting code for instruction I_i. For an instruction I_i, in order to evaluate the correctness of generated code C_i' on a LLMs. We first execute the Code C_i in DOCFORMEVAL, read the values of modified properties, obtaining a set $V_i = \{v_{i1}, v_{i2}, ..., v_{im}\}$. Next, we execute the generated code C_i' and obtain a set $V_i' = \{v_{i1}', v_{i2}', ..., v_{im}'\}$. Meanwhile, the runtime environment $\mathscr{E}$ outputs a feedback F_i. F_i is $true$ if C_i' runs without throwing exceptions, or F is $false$ if C_i' throws exceptions during execution. Based on those, we can calculate Formatting Accuracy (FA), the Formatting Error Rate (FER) and Execution Exception Rate (EER) as followed:

$$FA = \sum_{i=1}^{n} \left(\frac{\mathbb{I}(F_i = true)}{n} \sum_{j=1}^{m_i} \frac{\mathbb{I}(v_{ij} = v_{ij}')}{m_i}\right) \quad (4)$$

$$FER = \sum_{i=1}^{n} \left(\frac{\mathbb{I}(F_i = true)}{n} \sum_{j=1}^{m_i} \frac{\mathbb{I}(v_{ij} \neq v_{ij}')}{m_i}\right) \quad (5)$$

$$EER = \sum_{i=1}^{n} \frac{\mathbb{I}(F_i = false)}{n} \quad (6)$$

$$FA + FER + EER = 100\% \quad (7)$$

A.5 Detailed Hyperparameter Settings

For a stable output, we set the temperature to 0. Since some models, e.g., Qwen2 and CodeQwen, do not support setting the temperature to 0, we set their temperatures to 0.001. We also explore how different baselines perform in zero-shot and few-shot settings. In the zero-shot setting, for the DOC PROMPTING, we search for the optimal number of input knowledge pieces between 5 and 20, which we eventually set to 20. In the few-shot setting, we search between 1 and 21 for the optimal number of demonstrations. Finally, the shot number for both PROMPTING$_{few\text{-}shot}$ and DOC PROMPTING$_{few\text{-}shot}$ is set to 17. DOC PROMPTING$_{few\text{-}shot}$ also takes 15 pieces of knowledge as input. We set the maximum number of iterations to 3 for all baselines that employ the SELF-REFINEMENT mechanism.

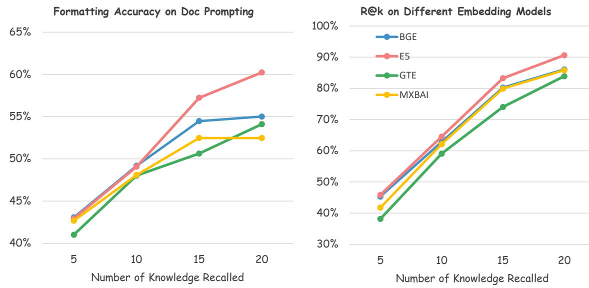

Figure 11: Impact of formatting accuracy and recall accuracy(R@k) by the number of knowledge recalled.

A.6 Effect of Retrieval Performance

We further explore the impact of different embedding models for vector retrieval on Text-to-DocForm to guide readers with different model selection needs. Specifically, to avoid introducing too many variables, we adopt TEXT-TO-FORMAT without the self-refine mechanism based on the Doc prompting strategy. We conduct experiments using four embedded models, respectively. The results are illustrated in Figure 11. As can be seen, the retrieval accuracy and TEXT-TO-FORMAT execution accuracy both improve as the number of recalled knowledge increases. Their performance is highly correlated. The E5 embedding model outperforms the other models and is recommended for vectorizing formatting instructions and related knowledge fragments. We consider the E5 model to be more suitable for the retrieval of API knowledge. In consequence, we choose $E5$ as retrieval model in TEXT-TO-FORMAT.

A.7 Effect of the Number of Shots

We conduct experiments to understand why few-shot learning can improve TEXT-TO-FORMAT execution accuracy. Specifically, we adopt the TEXT-TO-FORMAT based on the PROMPTING$_{few\text{-}shot}$ strategy as our baseline, removing the SELF-REFINEMENT mechanism. The shot number N specifies the value of $\{1,5,9,13,17,21.\}$. Constrained by the costs, we choose Gemini Pro and GPT-3.5 Turbo as closed-source models and DeepSeek-Chat and Qwen2 as open-source models. The experimental results are pictured in Figure 12. With the increase in the number of shots, the performance shows a trend of initially rising, reaching its peak at 17 shots, and then declining. In addition, it seems that the execution accuracy is mainly affected by the exception thrown, while the formatting error rate remains relatively stable. This indicates that the demonstrations provided mainly help to reduce exceptions in generated code, thereby improving execution accuracy.

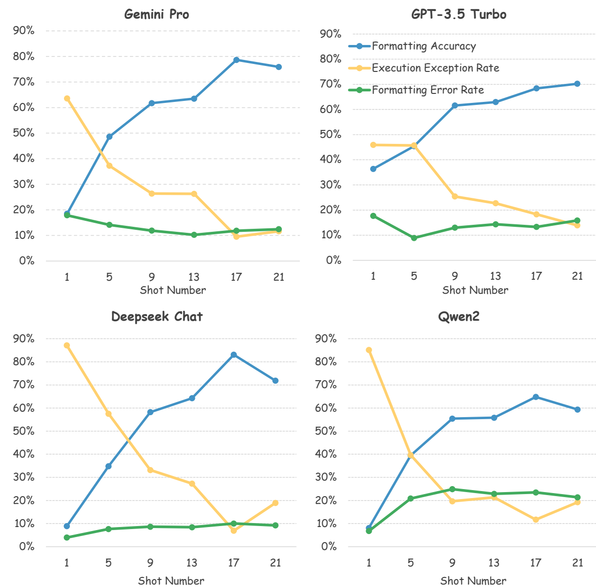

Figure 12: Impact of performance by number of shots.

A.8 Effect of Different Shots

We conduct an experiment to explore the impact of different shots on Execution Accuracy. We take the few-shot prompting without self-refinement strategy as the **Baseline**, corresponding to the results for Deepseek-Chat and Qwen2-7B shown in Figure 11. We set the number of shots to $\{1,5,9,13,17,21.\}$. In terms of demonstration selection for few-shot, we design two random selection strategies. The first strategy is named **One Random**. It randomly selects a specified number of demonstrations to construct the prompt, and then, all test examples are tested with the same demonstrations. That is, the prompting demonstrations are randomly chosen only once when conducting different n-shot experiments. The second strategy is called **Multiple Random**, which differs from **One Random** in that the demonstrations in the prompt are randomly selected again when testing different examples. We construct a selection pool with 50 demonstrations as a source of demonstration sampling.

We choose Deepseek-Chat (236B) and Qwen2-7B as representatives of large-scale and small-scale LLMs, respectively, for our experiments. The experimental results are summarized in Figure 13. It can be observed that the selection strategy of the demonstrations in the prompts can significantly affect the performance, and the number of demon-

strations required to achieve optimal performance varies for different selection strategies. We consider that this non-robustness may be because neither Deepseek-Chat nor Qwen2-7B models have learned the document formatting task during the training phase. This also points to the direction for the subsequent optimization of the TEXT-TO-FORMAT method, i.e., we can reduce token consumption by investigating how to select the optimal prompting demonstrations.

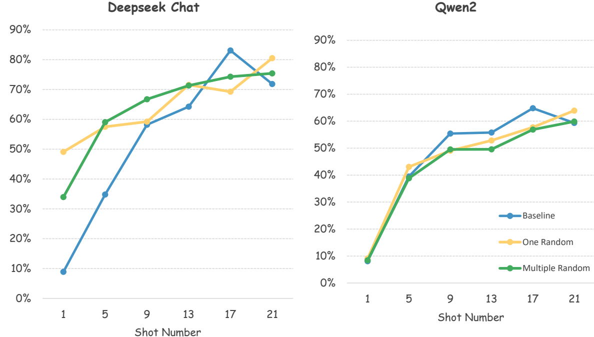

Figure 13: Impact of different shots.