Comments as Natural Logic Pivots: Improve Code Generation via Comment Perspective

Yijie Chen[1*], **Yijin Liu**[2], **Fandong Meng**[2], **Yufeng Chen**[1], **Jinan Xu**[1†], **Jie Zhou**[2]
[1]Beijing Key Lab of Traffic Data Analysis and Mining,
Beijing Jiaotong University, Beijing, China
[2]Pattern Recognition Center, WeChat AI, Tencent Inc, China
{22120354, chenyf, jaxu}@bjtu.edu.cn
{yijinliu, fandongmeng, withtomzhou}@tencent.com

Abstract

Code generation aims to understand the problem description and generate corresponding code snippets, where existing works generally decompose such complex tasks into intermediate steps by prompting strategies, such as Chain-of-Thought and its variants. While these studies have achieved some success, their effectiveness is highly dependent on the capabilities of advanced Large Language Models (LLMs) such as GPT-4, particularly in terms of API calls, which significantly limits their practical applicability. Consequently, enhancing the code generation capabilities of small and medium-scale code LLMs without significantly increasing training costs is an appealing challenge. In this paper, we suggest that code comments are the natural logic pivot between natural language and code language and propose using comments to boost the code generation ability of code LLMs. Concretely, we propose MANGO (comMents As Natural loGic pivOts), including a comment contrastive training strategy and a corresponding logical comment decoding strategy. Experiments are performed on HumanEval and MBPP, utilizing StarCoder and WizardCoder as backbone models and encompassing model parameter sizes between 3B and 7B. The results indicate that MANGO significantly improves code pass rate based on the strong baselines. Meanwhile, the robustness of the logical comment decoding strategy is notably higher than the Chain-of-thoughts prompting. [1]

1 Introduction

Recently, techniques on pre-trained Code Large Language Models (Code LLMs) are rapidly developing (Zhao et al., 2023) and perform well on various code-related tasks (Roziere et al., 2023; Chen et al., 2021). Code generation is to generate code snippets through prompting approaches for the given natural language description and optional extra information (*e.g,* data type, function name, and unit test examples). Additionally, the natural language description usually contains complex logic, making direct prompting difficult to solve hard problems. Decomposing complex problems into easier intermediate steps via the Chain-of-Thought (Wei et al., 2022) prompting strategy has shown promising performance in general complex tasks. For code generation, the follow-up works divide the planning and code synthesis process into two stages with uni-task few-shot prompts (Jiang et al., 2023) or intermediate steps with tree structure (Zelikman et al., 2023). In addition, constrain-

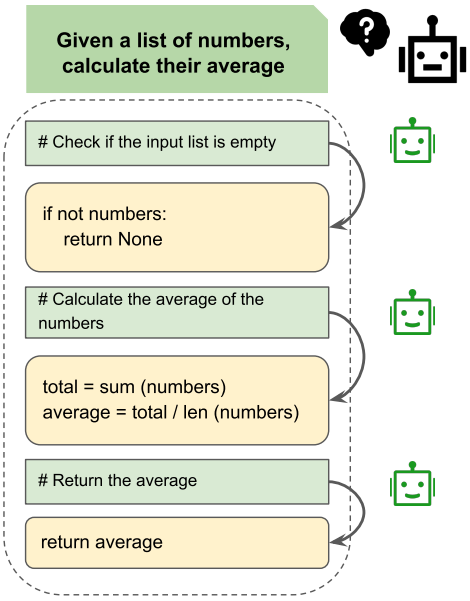

Figure 1: An illustration of the function of code-comment in code LLMs. The comment contributes to breaking down intermediate steps that correspond to the problem description and form an aligned structure with adjacent code lines.

[*] Work done when Yijie was interning at Pattern Recognition Center, WeChat AI, Tencent Inc, China.
[†] Jinan Xu is the corresponding author.
[1]The code is publicly available at https://github.com/pppa2019/Mango.

Findings of the Association for Computational Linguistics ACL 2024, pages 7040–7051

ing the intermediate steps using code structure information can also further improve the coding ability (Li et al., 2023a). However, task decomposing and planning are high-level and complex tasks that depend on the most advanced ability of LLMs (Zelikman et al., 2023). Consequently, enhancing the code generation of small to medium-sized LLMs efficiently presents a compelling and challenging issue.

To address the above issue, we propose the idea for the first time that code comments serve as the inherent logical pivot bridging natural language and programming language. Generally, comments within the code are commonly integral to the code corpus. Consequently, during the pre-training stage, training on code corpus endows the pre-trained code models with the respective capacities for understanding and generating code comments. As depicted in Figure 1, code comments can decompose the problem description using natural language, and each comment line establishes an alignment with the neighboring lines of code. Therefore, we hypothesize that encouraging models to generate comments can easily and effectively bridge the code and complex problem descriptions. In order to test our hypothesis, we discuss the possible inserting position of intermediate steps and use cross-entropy loss to evaluate the difficulty of various inserting style codes. According to the statistical results, we found that code in comment style is the easiest for various code models, which provides evidence for the hypothesis.

Based on the above hypothesis, we propose comMents As Natural loGic pivOts (MANGO) for code generation with problem description decomposition via comments. MANGO includes a logical comment decoding strategy and comment contrastive learning loss. Specifically, in the training phase, we generate negative samples without comments using the code data with comments in open-source datasets to strengthen the model preference for code with comments. During the decoding stage, the logical comment decoding strategy is adopted to guide the model in explaining the code logic via inline comments.

We conduct experiments on HumanEval and MBPP test sets using three backbone models from 3B to 7B. The experimental result shows that MANGO improves the pass rate consistently, *e.g.,* up to 7.52 pass@10 on HumanEval for StarCoder-7B and up to 4.17 pass@10 on MBPP for WizardCoder-7B. Our ablation studies show that each component of the method positively contributes to the performance of models, and the method is robust on the hyper-parameter and the various logical comment prompting styles. Furthermore, MANGO keeps consistent effectiveness in small-size models (*e.g.,* with pass@10 improvements of up to 3.87 on 3B and 4.07 on 7B), while CoT prompting can lead to severe decline, especially in 3B models. The error distribution and code feature data statistics are also provided for fine-grained analysis.

In summary, our contributions are as follows[2]:

- We first present that comments are pivots bridging natural language and code language, and conduct an analysis on comparing the difficulty of different positions of inserted decomposition steps for complex problem descriptions. To the best of our knowledge, this is the first work that fully explores the significant advantages of code-comments for coding problem decomposition.
- We propose MANGO that includes the contrastive training method and comment prompting strategy. MANGO improved the code generation ability of the models by strengthening the preference for code with comments and encouraging the model to use comments.
- The comparison between the CoT prompting strategy and our method indicates the effectiveness of code comments on small and middle model sizes. We conducted robustness evaluations on CoT and logical comment prompting strategies, respectively, and found that LCP achieves a much better performance with smaller standard deviations compared with CoT prompting.

2 Preliminary Analysis: Token-level Loss of Comment Related Code Styles

Braking difficult problems into easier intermediate steps benefits the code generation ability of LLMs in many existing works especially for API-based close-source models. However, most of the task decomposition methods can work only on the most advanced models like GPT-4, and their applicability to open-source models with small or middle sizes is limited. Therefore, we hypothesize that

[2]We uploaded the code as support material for review and it will be released on Github upon publication.

lowering the method difficulty of decomposition methods can benefit lowering the requirement of the model ability, and using comment is one of the possible methods. We define the possible styles for code with comment explanations and quantify their difficulties to models in zero-shot settings. First, we utilize WizardCoder-3B and WizardCoder-7B as backbone models, and the open-source code generation instruction tuning dataset CodeM as an evaluation dataset. We filtered the data with comments, extracted the comment line, and transferred it to the chain of thoughts. Then, given a task decomposition step is T_i and the corresponding code block is C_i (where $1 \leq i \leq n$, and n is the code block number), we construct three types of code, including the code with the preceding chain of thoughts (CoT-pre, which can be presented as $\{T_1, ..., T_n, C_1, ..., C_n\}$), the following chain of thoughts (CoT-post, which can be presented as $\{C_1, ..., C_n, T_1, ..., T_n\}$), and the inline comments (Comment, which can be presented as $\{T_1, C_1, ..., T_n, C_n\}$). The transferred CoT-style datasets contain equivalent information compared with the origin comment-style dataset.

The token-level cross-entropy loss can reflect the degree to which the model fits the specific token with the given context. Assuming the evaluated token is t_i, and the given context is $\mathbf{t_{<i}} = \{t_{i-1}, \cdots, t_0\}$, the token-level cross-entropy of the token t_i is $L(t_i)$. The $L(t_i)$ can be calculated as Equation 1.

$$L(t_i) = -log(exp(P(t_i|\mathbf{t_{<i}}))) \quad (1)$$

Using the token-level cross-entropy loss, we calculate the mean loss of the three datasets in different size models, including 3B and 7B. As shown in Figure 2 comments contain equivalent information with the lowest difficulty. We analyzed the output codes and found that the loss scores of natural language tokens are usually higher than those of programming language tokens, which is also mentioned in Zhu et al. (2023). However, when the intermediate steps of natural language are inserted in code lines as comments, their loss score decreases obviously. In Table 1, we observe that comment loss was significantly much lower than different CoT data, and the gap between cot-style and comment-style is larger when the model size is smaller. The observations above support our hypothesis that the code containing logical steps as comments lowers decomposition task difficulties for the models (especially for smaller models).

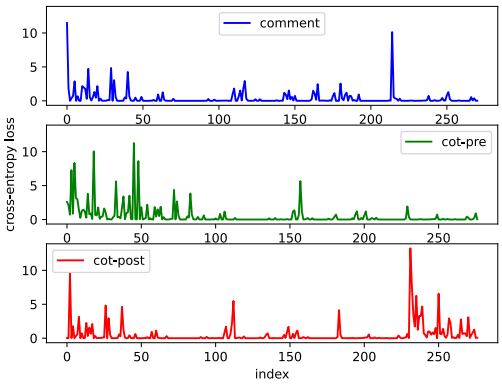

Figure 2: The token-level cross-entropy loss of three code styles for a code problem example in the WizardCoder3B model, and the abscissa represents the token index of the tokenized code.

Code Type	Wizardcoder 3b	Wizardcoder 7b
CoT-pre	0.5318	0.4784
CoT-post	0.5606	0.5241
Comment	0.3384	0.3086

Table 1: Average cross-entropy loss for three different code styles: code preceded with step list (CoT-pre), code followed with step list (CoT-post), and inline comments.

3 MANGO

We proposed a simple and effective method MANGO, including comment contrastive learning loss and logical comment prompt.

3.1 Backgound: Supervised Fine-tuning on Code Generation Tasks

The input of a code generation task typically involves a natural language description and an optional programming context. We denote such input as a list of tokens $\mathbf{x} = \{x_1, x_2, \cdots, x_n\}$. Given x, the correspondent code snippet $\mathbf{y} = \{y_1, y_2, \cdots, y_m\}$ is expected to be generated by the code LLM $P(\mathbf{y}|\mathbf{x})$. In standard supervised fine-tuning, the code LLMs are trained to predict the next token based on cross-entropy loss L_{lm} as Equation 2.

$$L_{lm} = -\sum_{i}^{m} logP(y_i|y_{i-1}, \cdots, y_1, \mathbf{x}) \quad (2)$$

3.2 Comment Contrastive Learning

In code snippets, code comments naturally assume the role of interpreting local code lines using natural language text, where the decomposition of

logic also serves as a function of comments. However, directly fine-tuning the loss function treats all tokens as equivalent without making additional distinctions for comments. To accommodate the task of guiding the model to generate annotations, we adopt a contrastive learning approach to encourage the model to emphasize code comments more during the fine-tuning process.

Our approach consists of the following steps. First, to prepare data for comment contrastive learning, we filter out examples containing comments from the training data. Taking the open-source dataset Code-Python as an example, this dataset contains half of the examples with annotations. For a code snippet $\mathbf{y}_{pos}$ containing comments, we use an open-source code parsing tool[3] to remove the comments and obtain the non-preferred contrastive sample $\mathbf{y}_{neg}$. Then, we add a contrastive loss L_{cl} by setting a margin m for the possibility of the labels with comments $\hat{\mathbf{y}}_{pos}$ and without comments $\hat{\mathbf{y}}_{neg}$.

$$L_{cl} = max(0, m - \log P_\theta(\hat{\mathbf{y}}_{neg}|\mathbf{x}) + \log P_\theta(\hat{\mathbf{y}}_{pos}|\mathbf{x})) \quad (3)$$

The final loss L is the addition of the standard cross entropy loss L_{lm} and the contrastive loss L_{cl}.

$$L = L_{lm} + L_{cl} \quad (4)$$

3.3 LCP: Logical Comment Prompt

In order to enable the model to use comments as intermediate steps intentionally, adding a corresponding instruction in the prompt is the method with the minimum cost. We use the logical comment prompting strategy to guide the model in generating comments explaining the code logic in the decoding stage. Following the standard prompt including the problem description and instruction for generating code in a certain programming language, the logical comment prompt is shown in Table 2, which adds "*with comments to explain the logic:*" or similar text.

4 Experiments and Results

In this section, we conduct experiments for our method on different benchmarks and backbone models, and the following sections describe the details of the experiments.

[3]https://github.com/pygments/pygments

4.1 Trainning Settings

We selected the state of the arts open-source backbone WizardCoder and StarCoder from 3B to 7B. We use **codem-python** (Zan et al., 2023) as training data, which includes 9600 Python examples distilled from GPT-4 using Evol-instruct (Xu et al., 2023). Our training script mainly follows WizardCoder (Luo et al., 2023) and trains 3 epochs (111 steps) with batch size 256; the warmup step is 15. For the hyperparameter margin m in our comment contrastive training loss, we set $m = 0.1$ for all models in the experiments. We use DeepSpeed[4] Stage 1 for distribution training. Our standard supervised fine-tuning implementation results have an error margin within 1% the Pass@1 result in the greedy search strategy in CodeM-Python (Zan et al., 2023), which indicates that our experiment setting leads to consistent results compared with the existing work.

SFT (Supervised Fine-Tuning) We construct the standard instruction template in WizardCoder (Luo et al., 2023) for the CodeM-Python dataset, and the training objection is cross entropy.

CoT (Chain-of-Thought) As another baseline, this setting uses zero-shot-CoT (Kojima et al., 2022) on the standardly fine-tuned models, and the detailed prompt is the "CoT-pre1" appended in the Appendix C Table 13.

LCP (Logical Comment Prompt) This setting uses LCP to guide standard fine-tuned models generating codes with comments. Table 2 shows the detailed prompt used in experiments.

MANGO (comMents As Natural loGic pivOts) During the training stage, MANGO trains the same dataset in the same instruction template, and the training objection is comment contrastive loss following Equation 3 and Equation 4. During the inference stage, the logical comment prompt (LCP) strategy will be used.

Decoding Settings We follow the setting in CodeLLaMA (Roziere et al., 2023), using temperature $T = 0.8$ and nuclear sampling $top\_p = 0.95$. We generate n = 10 samples and calculate Pass@k, where k = 1, 5, 10.

4.2 Evaluation Settings

Evaluate Metrics Pass@k (Chen et al., 2021) is currently the most widely used metric in code

[4]https://github.com/microsoft/DeepSpeed

Prompt Type	Content
Standard	### Instruction: Create a Python script for this problem: {input} ### Response: Here's the Python script for the given problem:
LCP	### Instruction: Create a Python script for this problem: {input} ### Response: Here's the Python script for the given problem with comments to explain the logic:

Table 2: The standard instruction-following prompt and LCP (Logical Comment Prompt), and the blue text guide the models to generate code with comments.

setting	HumanEval			MBPP		
	Pass@1	Pass@5	Pass@10	Pass@1	Pass@5	Pass@10
StarCoder-7B						
SFT	35.93	55.69	63.01	42.79	56.41	61.13
CoT	22.40	49.39	58.95	26.45	52.27	59.87
LCP	39.17	61.91	70.12	**41.11**	**57.68**	63.27
MANGO	**39.73**	**62.81**	**70.53**	40.92	**57.68**	**63.53**
WizardCoder-3B						
SFT	33.37	51.43	58.12	35.98	51.51	56.80
CoT	17.28	41.09	51.83	21.38	44.89	52.20
LCP	34.17	**56.07**	**63.82**	34.79	51.23	57.47
MANGO	**34.47**	54.55	61.99	**35.53**	**52.04**	**58.27**
WizardCoder-7B						
SFT	52.80	69.74	74.59	42.73	56.77	62.33
COT	42.50	68.68	75.20	31.25	52.65	59.67
LCP	52.95	70.70	75.81	41.96	57.82	63.27
MANGO	**54.59**	**72.76**	**78.66**	**46.91**	**62.26**	**67.87**

Table 3: The main experiments on MANGO. The **bold** text is the best result in experiments or comparable settings.

generation. It means with sampling n samples, the possibility of any code is correct in k samples.

$$\text{Pass@}k = \mathbb{E}_{\text{Problems}}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right] \quad (5)$$

HumanEval HumanEval (Chen et al., 2021) is currently the most widely used dataset for code generation. It contains 164 problems written in Python that evaluate programming language comprehension, algorithms, and basic mathematics. For each problem, 7.7 unit tests are contained on average.

MBPP MBPP (Austin et al., 2021) test set contains 500 Python problems. The problems cover basic programming concepts and standard library functionality. Each problem comprises a task description, code solution, and 3 automated test cases.

4.3 Main Result

In assessing the efficacy of MANGO, we utilize pass rates on HumanEval and MBPP as our primary evaluative measure. The principal results are presented in Table 3. Overall, there are three main aspects that we observed.

First, comparing the SFT baseline with MANGO, consistent enhancements are demonstrated across various sizes and backbones. Notably, the StarCoder-7B exhibits a significant augmentation, with an increase of 7.52 percentage points for HumanEval in Pass@10. In the case of WizardCoder-7B, there is a 4.17 percentage points enhancement

in the Pass@10 outcome for the MBPP test set. Secondly, comparing the pass rates of the backbones in the CoT settings, we observe a decline in model performance in all instances, with a more pronounced gap evident in the smaller 3B model. In contrast, LCP prompting consistently contributes to enhancing model performance, demonstrating its stability and effectiveness. Thirdly, in the ablation of LCP and MANGO, we observed that on the HumanEval and MBPP test set, a standardly finetuned StarCoder-7B can achieve a significantly higher score simply by prompting with LCP. This observation underscores the substantial potential of low-cost prompting methods. For deeper and broader analysis, we also provide the inner representation analysis on comment contrastive learning in Section A and the results of LCP on three larger and latest backbone models in Section B.

Considering the convenience of providing comparable results on zero-shot settings, we also provide the greedy search results of Pass@1 for comparison with the results of existing works. Since Pass@1 is the most strict indicator, it is more difficult to improve than Pass@10. According to results in Table 4, the zero-shot CoT prompting persistently results in diminished model performance; however, an enhancement in Pass@1 is observable when employing LCP prompting.

Model	Pass@1	Pass@5	Pass@10
GPT-4	67.00*	–	–
CodeLLaMA-7B	38.40*	–	–
StarCoder-7B	26.83*	–	–
WizardCoder-7B	55.50*	76.03	82.31
WizardCoder-3B	34.80*	51.72	59.75
w/ CoT	25.61	40.05	48.78
w/ LCP	36.59	51.57	59.15

Table 4: Zero-shot performance of various backbones on HumanEval. The Pass@1 is in greedy search decoding and the Pass@5/10 is in sampling decoding with T=0.8. We use * to denote the results from Achiam et al. (2023), Roziere et al. (2023), Li et al. (2023b) and Luo et al. (2023).

4.4 Ablation Study on the Training Method

We evaluate the efficacy of the two components of MANGO through an ablation study conducted on WizardCoder-7B. The test set employed is HumanEval, and we adhere to the main experimental settings with T=0.8 and p=0.95. Based on the results presented in Table 5 and Table 3, both the comment prompt and the contrastive training loss contribute to an increase in the Pass@10 passing rate. Furthermore, integrating these two components can further augment the performance in terms of the passing rate.

Setting	Pass@1	Pass@5	Pass@10
SFT	52.80	69.74	74.59
w/ LCP	52.95	70.70	75.81
w/ L_{cl}	**55.47**	71.75	76.02
MANGO	54.59	**72.76**	**78.66**

Table 5: Ablation study on the contrastive loss of MANGO based on WizardCode-7B.

4.5 The Effect of Margin

We marginalize the representation of code without comments with code LLMs, and the hyperparameter margin m is used to control the extent of negative marginalization. We examine various margin settings for WizardCoder-7B on HumanEval and follow decoding setting T=0.8. The results in Table 6 indicate that MANGO can outperform the baselines under margins larger than 0.05. However, when the margin is 0.03, the performance of the model decreases significantly both on Pass@5 and Pass@10, which indicates the model is sensitive to the margin value.

Margin	Pass@1	Pass@5	Pass@10
0.03	53.11	70.42	75.20
0.05	53.66	72.98	78.66
0.10	54.59	72.76	78.66
0.15	53.13	73.00	79.88

Table 6: Roubustness validation on the margin hyperparameter m.

5 Analysis

5.1 The Robustness Study Against LCP and CoT

We rephrase the prompt into several versions to mitigate the model's randomness in response to different prompt styles. Initially, we manually created three prompts each for LCP and CoT, which served as seed prompts. Subsequently, we employ GPT-4 to generate four additional variants for each prompt. Furthermore, for a more nuanced analysis, we segregate CoT into two categories: the first involves specifying the chain of thought text

before generating the final code, representing the most typical and universal form; and the second is a freestyle CoT devoid of position-constraining guidance. Ultimately, we have 15 prompts for each category. The complete prompts are provided in the Appendix 5.1. The result in Table 7 demonstrated that LCP yields a considerably higher average performance and a lower standard deviation. It is worth noting that the CoT without position constraint has higher performance than typical CoT on average with a larger deviation. Apart from the most commonly used independent natural language plan before actual coding, the model can also interpret CoT as explanatory comments for the code. LCP proposes that intermediate steps should be incorporated as comments, thereby stabilizing the behavior of models and enhancing their performance compared to generating code problems directly.

Setting	Pass@1	Pass@5	Pass@10
CoT-Pre			
Avg	33.24	61.88	71.38
StDEV	9.21	6.45	5.43
LCP			
Avg	48.65	68.85	74.76
StDEV	1.37	1.74	2.45
CoT-No-Position			
Avg	40.69	65.05	72.03
StDEV	10.64	7.19	5.46

Table 7: The mean and standard deviation of three prompt groups. CoT-Pre means the prompts request models that generate the Chain-of-Thought first and then generate code, while CoT-No-Position means the prompts guide models generate code with Chain-of-Thought only.

5.2 Statistical Features of Generated Codes

We statistic the style transformation between the different instruction strategies and training methods, including the effective code line number and comment line number. As Tabel 8, the CoT prompt guides model generates intermedia steps before generating code, leading to fewer comment lines than the prompts without guidance. LCP induces more code comments than the origin prompt.

We observe that the contrastive training loss L_{cl} does not change the code and comment line number features. However, in model output without contrastive learning, 12.21% of comment lines are " Test the function" (or similar expressions) and followed with some unit tests, which will not help the code's correctness. Conversely, a mere 2.05% of comment lines exhibit this characteristic after training by comment contrastive learning. The above statistical results indicate that, from an effective comment ratio perspective, comment quality is enhanced after training with contrastive learning.

setting	#avg. code line	#avg. comment line
origin	15.56	1.36
CoT	11.68	0.38
LCP	20.92	3.29
MANGO	20.80	3.33

Table 8: Statistic of code and comment line on different settings based on WizardCoder-7B.

5.3 The Relationship Between CoT and Contrastive Comment Loss

To investigate whether the enhancement of comparative learning for comments can be directly generalized to the capability of CoT. Given the substantial fluctuation in the code generated by CoT's prompt, we selected three typical prompts as the study subjects. Through the statistical analysis of the number of lines in the generated code, these three prompts respectively represented code styles with three different mean values of 0, 2, and 3 on WizardCoder-7B, and the detailed prompt will be listed in the Appendix (CoT1: CoT-No-Position1, CoT2: CoT-pre1, CoT3: CoT-pre2). We test the model in the SFT setting and the contrastive trained setting. The results are shown in Table. 9. Comparing the mean of comment lines and the final pass rates, we can find that the more comments the CoT prompt can guide, the higher performance can be seen. Comparing SFT and L_{cl} under the same settings, it can be observed that for the prompt that inherently performs CoT through code comments (i.e., "CoT1"), the models after contrastive learning consistently improve code pass rates. However, for prompts that perform CoT through other means, a decline in performance is observed in the 3B models after comparative learning. This phenomenon suggests that merely fine-tuning through a small amount of contrastive learning on comments is not easily transferable to other forms of task decomposition.

setting		pass@1	pass@5	pass@10	MCL
		WizardCoder-3B			
CoT1	SFT	28.80	52.22	61.38	2.83
	L_{cl}	30.10	52.85	60.77	2.64
CoT2	SFT	17.91	44.05	53.86	1.35
	L_{cl}	15.65	40.84	52.03	1.06
CoT3	SFT	17.28	41.09	51.83	0.95
	L_{cl}	12.85	34.97	45.33	1.03
		WizardCoder-7B			
CoT1	SFT	47.64	70.89	77.64	2.70
	L_{cl}	50.43	71.68	77.64	3.03
CoT2	SFT	47.70	67.94	73.78	1.86
	L_{cl}	44.80	70.28	76.42	1.89
CoT3	SFT	42.30	67.76	74.19	0.38
	L_{cl}	41.02	68.31	75.81	0.26

Table 9: The relevance of the CoT prompting performance and the training loss setting. "MCL" is the abbreviation of mean comment lines.

5.4 Error Distribution

To analyze the pass rate improvement of MANGO, we classify problem description understanding error types using the HumanEval test set. MANGO improves model output by utilizing code comments, which does not target a specific type of code error but rather improves the understanding of problem descriptions from the overall perspective via decomposing complex coding logic. Meanwhile, test cases can effectively reflect the degree of understanding of the output results in the code generation task. We established two thresholds for evaluating problem understanding: whether the compiler can successfully compile the code and whether the code can pass at least one test case. Based on this, we roughly classified the understanding level into three types from low to high.

We divide the failed cases into three categories: **Verification Error (VE)** includes Runtime Error, Syntax Error, and the other case except for Wrong Answer; **All Wrong Answer (AWA)** includes the cases that cannot pass the first test cases of the scripts, indicating that the generated code has a poor understanding on problem descriptions; **Particle Wrong Answer (PWA)** includes the cases that can pass the first unit test at least but fail finally.

As shown in Tabel 10, compared with the error rates in SFT, MANGO decreases all three types of error rates. Furthermore, the ablation components also show lower error rates in three types, indicating that MANGO has reduced the code error rate across various degrees of comprehension error.

Setting	VE	AWA	PWA	overall
SFT	4.31	10.24	11.87	26.42
w/ LCP	3.84	8.88	11.46	24.19
w/ L_{cl}	**2.81**	10.69	10.49	23.99
MANGO	2.84	**8.49**	**10.00**	**21.33**

Table 10: Statistics of test result error distribution.

6 Related Work

6.1 Code LLMs and Logical Reasoning

The rapid development of pre-trained language models brings the bloom of code LLMs. The most advanced open-source code LLMs are pre-trained by a large amount of natural language corpus and code corpus (Roziere et al., 2023). Based on the backbones, WizardCoder (Luo et al., 2023) and OctoPack (Muennighoff et al., 2023) proposed instruction-tuning data construction methods that boost the instruction-following ability of code LLMs significantly. Besides, Fu et al. (2022) proposed that code corpus can be the source of CoT ability. Both Fu et al. (2022) and Ma et al. (2023) observed that training small-size code LLMs on code data can augment the logical reasoning ability of the model further.

6.2 Prompting-based Code Generation Strategy

The planning strategy is the mainstream technique to transform the causal language model to fit logical text generation. Existing works include the variant of CoT (Chain of Thought) (Jiang et al., 2023; Li et al., 2023a), Reflexion (Shinn et al., 2023), etc. Most prompting strategies utilize the outstanding logical and understanding abilities of the most advanced models. However, these strategies will not work even on large open-source models with a size beyond 10B (Jiang et al., 2023; Shinn et al., 2023). Another mainstream method is compiling feedback to train code models (Chen et al., 2023, 2022). Our method utilizes the comment, the widely accepted rule for coders, to enhance the code readability and explain the code logic simultaneously. Li et al. (2023c) proposed that the brainstorming step can elite code generation performance. The commonness of comments in training code corpus enables models to easily capture the relationship between code and natural language description logic in different sizes of backbone models.

7 Conclusion

In this paper, we first emphasize the importance of code comments in bridging natural language and code language, and then we hypothesize that more comments to explain code logic can enhance code generation performance. We propose a simple and effective method MANGO, which contains contrastive comment loss and logical comment prompts. The main experiments and ablation studies show that MANGO effectively augments the pass rate of code generated by the models. In further analysis, we also compared LCP and CoT prompting strategies and found that LCP has a significantly better and stabler performance than CoT, especially for smaller models. The comment contrastive training can also boost CoT performance in certain conditions.

In conclusion, our work introduces the comment pivot as a novel perspective, and guiding models to use comments can augment the code generation ability stably. In the future, we will further explore the application potential of comments in broader complex scenarios, such as tool usage and LLM agents.

8 Limitation

Although MANGO does not constrain the size of the model, we did not test our method (especially for comment contrastive training) on larger models due to our limitations in computing resources. Furthermore, the method adopts a strategy to stimulate the model's annotation capabilities, but this method is still limited by the upper limit of the planning and comment generation capabilities of the models. Therefore, exploring how to further improve the annotation capabilities of the base model is an issue worth exploring.

Acknowledgements

The research work described in this paper has been supported by the National Nature Science Foundation of China (No. 62376019, 61976015, 61976016, 61876198, and 61370130). The authors would like to thank the anonymous reviewers for their valuable comments and suggestions to improve this paper.

References

Marah Abdin, Sam Ade Jacobs, Ammar Ahmad Awan, Jyoti Aneja, Ahmed Awadallah, Hany Awadalla, Nguyen Bach, Amit Bahree, Arash Bakhtiari, Harkirat Behl, et al. 2024. Phi-3 technical report: A highly capable language model locally on your phone. *arXiv preprint arXiv:2404.14219*.

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*.

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 preprint arXiv:2108.07732*.

Angelica Chen, Jérémy Scheurer, Tomasz Korbak, Jon Ander Campos, Jun Shern Chan, Samuel R Bowman, Kyunghyun Cho, and Ethan Perez. 2023. Improving code generation by training with natural language feedback. *arXiv preprint arXiv:2303.16749*.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. Codet: Code generation with generated tests. In *The Eleventh International Conference on Learning Representations*.

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 preprint arXiv:2107.03374*.

Yao Fu, Hao Peng, and Tushar Khot. 2022. How does gpt obtain its ability? tracing emergent abilities of language models to their sources. *Yao Fu's Notion*.

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

Xue Jiang, Yihong Dong, Lecheng Wang, Qiwei Shang, and Ge Li. 2023. Self-planning code generation with large language model. *arXiv preprint arXiv:2303.06689*.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213.

Jia Li, Ge Li, Yongmin Li, and Zhi Jin. 2023a. Structured chain-of-thought prompting for code generation. *arXiv preprint arXiv:2305.06599*.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023b. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Xin-Ye Li, Jiang-Tian Xue, Zheng Xie, and Ming Li. 2023c. Think outside the code: Brainstorming boosts large language models in code generation. *arXiv preprint arXiv:2305.10679*.

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 preprint arXiv:2306.08568*.

Yingwei Ma, Yue Liu, Yue Yu, Yuanliang Zhang, Yu Jiang, Changjian Wang, and Shanshan Li. 2023. At which training stage does code data help llms reasoning? *arXiv preprint arXiv:2309.16298*.

Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. 2023. Octopack: Instruction tuning code large language models. *arXiv preprint arXiv:2308.07124*.

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*.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik R Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837.

Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, and Daxin Jiang. 2023. Wizardlm: Empowering large language models to follow complex instructions. *arXiv preprint arXiv:2304.12244*.

Daoguang Zan, Ailun Yu, Bo Shen, Jiaxin Zhang, Taihong Chen, Bing Geng, Bei Chen, Jichuan Ji, Yafen Yao, Yongji Wang, et al. 2023. Can programming languages boost each other via instruction tuning? *arXiv preprint arXiv:2308.16824*.

Eric Zelikman, Qian Huang, Gabriel Poesia, Noah Goodman, and Nick Haber. 2023. Parsel: Algorithmic reasoning with language models by composing decompositions. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. 2023. A survey of large language models. *arXiv preprint arXiv:2303.18223*.

Yuqi Zhu, Jia Allen Li, Ge Li, YunFei Zhao, Jia Li, Zhi Jin, and Hong Mei. 2023. Improving code generation by dynamic temperature sampling. *arXiv preprint arXiv:2309.02772*.

A Analysis on the Effect of Contrastive Learning

To show how comment contrastive learning changes the inner representation of the model, we compare the cosine similarity of representation for code snippets with comments and the unpreferred variations.

First, we sample 100 code snippets with comments from the training set and post-process the code snippets into two unpreferred code styles, including shuffling comments and deleting comments. Secondly, we process each code snippet into three versions listed below:

- Origin: The origin code retains its comments without change.
- Shuffling Comments (SC): The comments are shuffled without changing the inserting position, which breaks the semantic alignment relationship between the certain comment and its nearby code lines.
- Deleting Comments (DC): The comments are eliminated, leaving only the executable code lines.

Concretely, we compare the model using standard finetuning and comment contrastive learning by computing the cosine similarity of the original and the other versions, respectively. The cosine similarity score is calculated with the last token representation of the last layer model output. Finally, the final scores are averaged across the 100 samples. The cosine similarity scores are displayed in Table 11. The table shows that cosine similarity scores between the origin version and the other two unpreferred versions are substantially lower in the model with comment contrastive learning. This suggests that the contrastive learning process

model	<origin, SC>	<origin, DC>
SFT	0.7780	0.5271
w/ L_{cl}	0.7231	0.4404

Table 11: The cosine similarity score of different types of post-processing strategies, where SC means shuffling comments and DC means deleting comments.

markedly enhances their differentiation within the embedding space.

B Performance on Boarder Models

We evaluate our decoding strategy LCP to explore the potential of utilizing comments to improve code generation on larger and more variant models. We compare LCP on WizardCoder-13B (Luo et al., 2023), Phi-3-14B (Abdin et al., 2024), and DeepSeekCoder-7B (Guo et al., 2024). The Pass@1 results in temperature T=0.8 are shown in Table 12 are based on HumanEval, indicating that LCP consistently and significantly surpasses the origin setting. Moreover, CoT can promote the code generation on DeepSeekCoder-7B and Phi-3-14B, but it fails to improve the pass rate in WizardCoder-13B. Therefore, the results further illustrate the stability and effectiveness of LCP on border backbone models and reveal the potential of comments in enhancing code generation.

model	origin	CoT	LCP
WizardCoder-13B	0.6402	0.5914	0.6524
DeepSeekCoder-7B	0.5792	0.6036	0.6158
Phi-3-14B	0.5915	0.6768	0.6585

Table 12: Performance on three latest and larger backbone models. The results reflect that LCP is consistently effective.

C The Detail of Variant Prompts

We use three types of prompts to generate codes with different styles. The CoT-pre indicates prompts that request models to generate a step-by-step chain of thoughts before generating code, and the CoT-No-Position indicates prompts that only request models generate a step-by-step chain of thoughts. The concrete prompts of three types are shown as follows: We show the concrete prompts of the three seed prompts for each type in Table 13, and the other prompts are paraphrased by GPT-4.

D Case Study

We use a simple coding task to illustrate the difference between the origin prompt, the LCP prompt, and the CoT prompt. The problem description is shown as follows.

```
def is_simple_power(x, n):
    """Your task is to write a function that
        returns true if a number x is a simple
    power of n and false in other cases.
    x is a simple power of n if n**int=x
    For example:
    is_simple_power(1, 4) => true
    is_simple_power(2, 2) => true
    is_simple_power(8, 2) => true
    is_simple_power(3, 2) => false
    is_simple_power(3, 1) => false
    is_simple_power(5, 3) => false
    """
```

Origin Prompt result [Failed] The origin prompt contains no extra instructions, and the generated code fails to pass all of the unit tests because it can only solve the cases provided as examples.

```
```python
def is_simple_power(x, n):
    """Your task is to write a function that
        returns true if a number x is a simple
    power of n and false in other cases.
    x is a simple power of n if n**int=x
    For example:
    is_simple_power(1, 4) => true
    is_simple_power(2, 2) => true
    is_simple_power(8, 2) => true
    is_simple_power(3, 2) => false
    is_simple_power(3, 1) => false
    is_simple_power(5, 3) => false
    """
    if n == 1:
        return True
    elif x == n:
        return True
    elif x == n**2:
        return True
    elif x == n**3:
        return True
    else:
        return False
```

In this script, we define a function called
    `is_simple_power` that takes two parameters
    `x` and `n`. We check a few possible cases
    to determine if `x` is a simple power of
    `n`. If any of these cases are true, we
    return `True`, otherwise we return `False`
```

CoT result [Failed] The code generated in the CoT prompt setting contains a step-by-step explanation that simply decomposes the problem description into two bullet points. The final generated code fails due to the generated ambiguous reinterpreted bullet points, and there is a large gap between the code implementation and the problem description.

```
1. Define a function called `is_simple_power`
    that takes two parameters, `x` and `n`.
2. Inside the function, check if `x` is equal
```

prompt name	content
CoT-pre1	First, consider the process step by step, and then present the Python script:
CoT-pre2	Start by creating a step-by-step coding plan, then follow it with the Python script for the provided issue:
CoT-pre3	The following Python script addresses the specified issue and includes a detailed explanation at the start:
Comment1	Here is the annotated Python script that solves the specified issue:
Comment2	To clarify the logic, it's preferable to include annotations in the script. Here's the Python code for the given issue:
Comment3	The following Python script addresses the given problem, with complex logic broken down using straightforward comments:
CoT-No-Position1	Here's the Python code for the specified issue, broken down into logical steps:
CoT-No-Position2	The Python program for the aforementioned issue is presented here, with a step-by-step explanation:
CoT-No-Position3	Here's a Python script that tackles the given problem, accompanied by a comprehensive breakdown:

Table 13: Three handcraft seed prompts for each type.

````
    to `n**int(n)`. If it is, return `True`.
    Otherwise, return `False`.

Here is the Python script implementing the
    above steps:

```python
def is_simple_power(x, n):
    if x == n**int(n):
        return True
    else:
        return False
```

This function checks if a number `x` is a
    simple power of `n`. If `x` is equal to
    `n` raised to the power of `int(n)`, it
    returns `True`. Otherwise, it returns
    `False`.
````

LCP result [Passed] The code generated in the LCP prompt setting contains clear comments to explain the code logic and all of the unit tests are passed successfully.

````
```python
def is_simple_power(x, n):
    """
    Your task is to write a function that
        returns true if a number x is a simple
    power of n and false in other cases.
    x is a simple power of n if n**int=x
    For example:
    is_simple_power(1, 4) => true
    is_simple_power(2, 2) => true
    is_simple_power(8, 2) => true
    is_simple_power(3, 2) => false
    is_simple_power(3, 1) => false
    is_simple_power(5, 3) => false
    """
    # Check if x is a perfect power of n
    if n == 1:
        return x == 1

    # Check if x is a simple power of n
    if x % n == 0:
        return is_simple_power(x // n, n)

    # Check if x is a perfect power of n
    for i in range(2, x):
        if x == n ** i:
            return True

    # If x does not satisfy any of the above
        conditions, it is not a simple power of
        n
    return False
```

Now, you can call the function
    `is_simple_power` with the desired inputs
    to check if a number is a simple power of a
    given base.
}
````