Automated Prompt Generation for Code Intelligence: An Empirical study and Experience in WeChat

Kexing Ji[1], Shiyun Fu[1], Cuiyun Gao[1*], Yujia Chen[1], Zezhou Yang[2], Chaozheng Wang[2], Yuetang Deng[2]
[1]The Chinese University of Hong Kong, Hong Kong, China
[2]WeChat Group, Tencent Inc., Guangzhou, China
kexing1208@gmail.com, FuShiyun1@outlook.com, cuiyungao@outlook.com, yujiachenhit@gmail.com,
zezhouyang@tencent.com, adf111178@gmail.com, yuetangdeng@tencent.com

***Abstract*—Large Code Models (LCMs) have demonstrated potential in advancing various code intelligence tasks. However, their effectiveness can be greatly influenced by the quality of the prompts. Current prompt design strategies in code intelligence studies are mostly manually generated, which could be time-consuming and extremely rely on the base LCMs and tasks. Although automated prompt generation (APG) has been investigated in the natural language processing field, it has not attracted sufficient attention and been well explored in the code intelligence tasks. Considering the various tasks and black-box nature of LCMs faced by developers in practice, it is essential to automate the prompt generation process.**

To mitigate the gap, we empirically investigate the two important parts in APG, including Instruction Generation (IG) and Muti-Step Reasoning (MSR). The instruction generation part aims at providing a task-related description for instructing LCMs to effectively accomplish specific tasks; while the multi-step reasoning part aims at guiding LCMs to produce a series of logical steps before arriving at the final answer. For each part, we evaluate the widely-used APG methods on four open-source LCMs and three code intelligence tasks, i.e., code translation (PL-PL), code summarization (PL-NL) and API recommendation (NL-PL). Experimental results indicate that the two parts in APG can dramatically enhance the performance of the code intelligence tasks compared with the basic prompts. Based on the results, we further propose a novel APG approach by combining the best methods of the two studied parts of APG. Experiments show that the proposed APG approach achieves an average improvement of 28.38% with respect to CodeBLEU for the code translation, 58.11% in terms of ROUGE-L for the code summarization and 84.53% in SuccessRate@1 for the API recommendation over the basic prompts, respectively. To validate the effectiveness in industrial scenario, we further evaluate our approach on WeChat-Bench, a proprietary dataset from the WeChat Group in Tencent for API recommendation, achieving an average improvement of 148.89% in MRR.

***Index Terms*—Automated prompting generation, Code intelligence, Large Code Models, Empirical study**

I. Introduction

Recently, the advent of large-scale code corpora combined with advances in deep learning technologies has facilitated the development of Large Code Models (LCMs), e.g., Code Llama [1], DeepSeek-Coder [2], and Qwen2.5-Coder [3]. These LCMs demonstrate great potential in advancing various code intelligence tasks, including code summarization [4], [5], API recommendation [6], [7] and code translation [8], [9]. To effectively employ LCMs in these tasks, users generally provide task-specific prompts that include both instructions and contextual information. Figure 1 provides an example of this interactive process. The user formulates a prompt containing the task description, source code, and reasoning steps for the code summarization task. Subsequently, the LCM receives the prompt as input and generates a response.

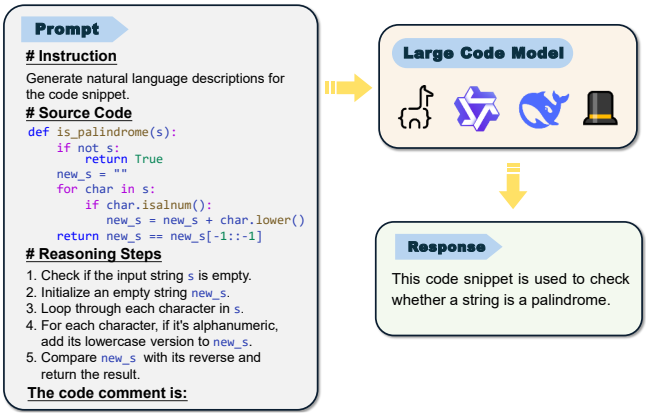

Fig. 1: An example of designed prompt for LCM to generate the summary of given code snippet.

Current prompt design strategies [10]–[12] in code intelligence are generally manually written. For instance, Sun *et al.* [13] manually design 11 different prompts and conduct experiments to decide which prompt is more suitable for specific LCMs to generate code comments. White *et al.* [14] manually craft 13 types of prompts for four task categories, including code quality improvement, refactoring, requirements acquisition, and software design. Although the human-written prompts achieve superior performance in code intelligence tasks, they bring several challenges. First, different human-written prompts can lead to dramatically different performance. For example, Sun *et al.* [13] report that the least effective prompt results in a 119.87% reduction in BLEU scores compared to the most effective prompt. Second, in industrial development scenarios, different projects often require task-specific and model-specific prompts tailored to

*Cuiyun Gao is the corresponding author.

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00285

their unique codebases, APIs, and development requirements, making manually-generated prompts hard to be reused across projects [15]. Moreover, the manual process is time-consuming and may bring much labor cost, particularly when scaling across various code intelligence tasks and LCMs. Therefore, how to automatically generate effective prompts for different tasks and LCMs becomes an important research question.

Recent studies in the natural language processing (NLP) field have proposed various Automated Prompting Generation (APG) methods [16]–[20] and have achieved promising results in NLP tasks. The APG methods leverage the task definitions and input data to automatically generate prompts for LCMs, enabling adaptability across different models and tasks. However, while APG methods have proven effective in the NLP field, the application of these methods to code intelligent tasks is still unexplored. Considering the importance of prompt design in code intelligence tasks, it is necessary to explore the APG methods for code intelligence tasks.

To this end, we conduct a comprehensive empirical study on the impact of APG for code intelligence tasks and evaluate its effectiveness in industrial scenario. Based on the constituent parts of a prompt, i.e., instruction and reasoning steps as illustrated in Fig. 1, we focus on two important parts in APG, including Instruction Generation (IG) and Multi-Step Reasoning (MSR) in this paper. Specifically, instruction generation aims at providing a task-related description for instructing LCMs to effectively accomplish specific tasks, while multi-step reasoning aims at guiding LCMs to produce a series of intermediate logical steps before arriving at the final answer.

For each aspect, we evaluate the widely-used APG methods on four open-source LCMs. We choose three popular code intelligence tasks, including API recommendation (NL-PL), code translation (PL-PL), and code summarization (PL-NL) for evaluation. We focus on investigating the following four research questions (RQs):

1) What is the impact of instruction generation on the effectiveness of APG?
2) How effective is multi-step reasoning in enhancing the performance of APG?
3) Can we further improve the performance of current APG methods?
4) How does the improved APG method perform in the industrial scenario?

To explore the impact of instruction generation on the effectiveness of APG methods (RQ1), we evaluate two representative automated instruction generation methods: Automatic Prompt Engineer [17] and Optimization by PROmpt (OPRO) [21], and compare them against manually crafted prompts [11], [22] across four open-source LCMs. To investigate the effectiveness of multi-step reasoning in enhancing the performance of LCMs (RQ2), we evaluate the three widely adopted multi-step reasoning methods—Chain-of-Thought (CoT) [23], AutoCoT [24], and Self-Plan [25] and compare their performance with that of manually designed prompts. To further explore whether the performance of current APG methods can be improved (RQ3), we combine the best-performing instruction generation and multi-step reasoning methods identified in RQ1 and RQ2, and evaluate them on four open-source LCMs for code intelligence tasks. Building upon the findings of RQ3, we further investigate whether the observed improvements in APG methods can generalize to the industrial scenario (RQ4), thereby evaluating the practical applicability of the proposed technique.

Based on the experimental results, we find that APG methods greatly improve LCM performance across three code intelligence tasks. Among the evaluated APG methods, APE proves to be the most effective instruction generation method, while CoT achieves the best results among multi-step reasoning methods. Furthermore, the novel approach APE-CoT achieve non-trivial performance improvements over basic prompts, with an average increase of 28.38% in CodeBLEU for code translation, 58.11% in ROUGE-L for code summarization and 84.53% in SucessRate@1 for API recommendation. Additionally, evaluation on WeChat-Bench, a proprietary dataset of 1000 C++ samples from the WeChat Group in Tencent for API recommendation, demonstrates an average improvement of 148.89% in MRR, confirming the effectiveness of our approach in the industrial scenario.

The main contributions of our work are summarized below:

1) To the best of our knowledge, this paper presents the first in-depth investigation into how to effectively utilize APG to generate prompts for code intelligence tasks.
2) Based on our findings, we propose a novel APG method that greatly improves the performance of LCMs in code intelligence tasks compared to the basic prompts.
3) We validate our approach in industrial scenario through evaluation in WeChat and offer practical implications for developers and potential directions for researchers.

II. Background

As shown in Fig. 1, APG contains two key parts: *instruction generation*, which creates a tailored task description based on the input, and *multi-step reasoning*, which produces structured steps to solve the task. We elaborate on the details of the two parts in the following.

A. Instruction Generation

For instruction generation, we consider two widely acknowledged and effective techniques, namely APE [17] and OPRO [21], as detailed below.

- **Automatic Prompt Engineer (APE) [17]** treats instruction generation as a natural language synthesis task. As shown in Figure 2, LCMs first automatically generate a pool of candidate instructions through comprehending the task requirements underneath basic prompt, and then select the best instruction through log probability. The log probability reflects the likelihood of producing the desired result given the instruction and the LCM.
- **Optimization by PROmpting (OPRO) [21]** uses LCMs to refine task execution through dynamic optimization. Figure 2 illustrates the OPRO process. A meta-prompt guides LCMs to generate and evaluate candidate instructions, and

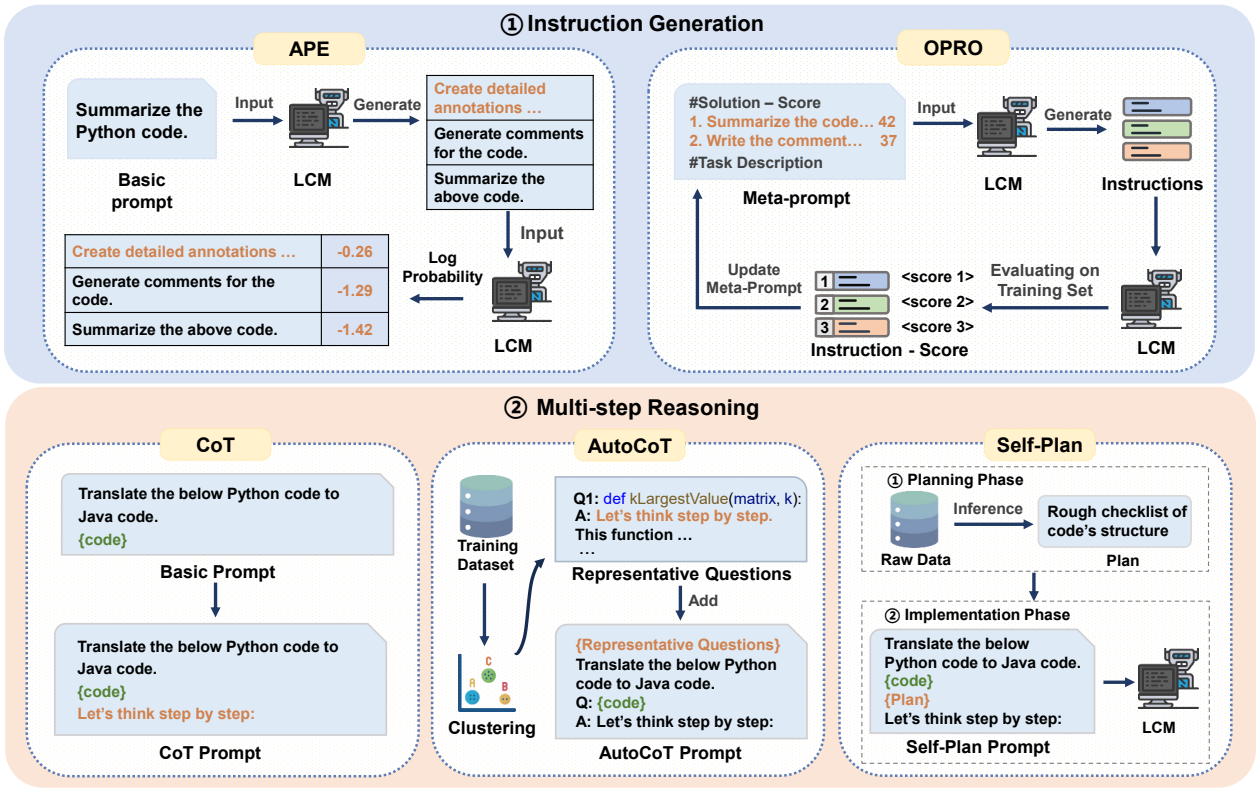

Fig. 2: The overview of APG Methods: Instruction Generation and Multi-step Reasoning.

the top-performing pairs are used to update the meta-prompt iteratively until convergence on an optimal instruction.

B. Multi-step Reasoning

For multi-step reasoning, we select three representative methods—CoT, AutoCoT, and Self-Plan—based on their remarkable performance in prior work [26], as detailed below.

- **Chain-of-Thought (CoT)** [23] enhances the reasoning capabilities of LCMs by guiding them through a sequence of intermediate steps, thereby reducing the difficulty of generating the desired output. For example, as shown in Figure 2, by simply adding "Let's think step by step", CoT prompt encourages LCMs to produce more considered responses by focusing on intermediary reasoning steps.
- **AutoCoT [24]** serves as an automated multi-step reasoning method that optimizes CoT in terms of efficiency. As shown in Figure 2, AutoCoT clusters the given training dataset and selects representative problems from each cluster. These representative questions, along with their reasoning steps, are then incorporated into the AutoCoT prompt to better guide LCMs in generating accurate outputs.
- **Self-Plan [25]** decomposes complex tasks into a structured subtasks, outlining the overall program structure without delving into implementation details. It provides clear and concise high-level actions through a two-stage generation process, as illustrated in Figure 2. In the first stage, the LCM generates a "plan" that serves as an outline for the code structure. In the second stage, the LCM leverages both the plan and code to generate outputs for downstream tasks.

III. EXPERIMENTAL SETUP

A. Research Questions

In this paper, we conduct experiments with the aim of answering the following research questions:

- **RQ1:** What is the impact of instruction generation on the effectiveness of APG?
- **RQ2:** How effective is multi-step reasoning in enhancing the performance of APG?
- **RQ3:** Can we further improve the performance of current APG methods?
- **RQ4:** How does the improved APG method perform in an industrial scenario?

B. Evaluation Tasks

We evaluate the APG approaches on three representative code intelligence tasks: code summarization, code translation and API recommendation.

1) Code Summarization: Code summarization, also known as code comment generation, targets to automatically generate concise natural language descriptions for source code snippets [27], [28]. We use the prompt template from [10] as our basic prompt:

TABLE I: Statistics of the datasets.

Task	Dataset	Train	Valid	Test
Code Summarization	CodeXGLUE-Java	164,923	5,183	10,955
	CodeXGLUE-Python	251,820	13,914	14,918
Code Translation	AVATAR-Java	60,138	476	1,906
	AVATAR-Python	60,138	476	1,906
API Recommendation	APIBENCH-Q-Java	4,769	594	600
	APIBENCH-Q-Python	3,118	391	400

"*Generate comments for* [LANG] *code.*"

where [LANG] denotes the slot of programming language.

Datasets. We use the CodeXGLUE code summarization benchmark [29]. It contains code snippets and corresponding natural language comments extracted from GitHub repositories. The detailed data statistics are shown in Table I.

Metrics. Following previous work [22], we use three widely adopted metrics for evaluating code summarization: BLEU-4 [30], ROUGE-L [31], and METEOR [32]. These metrics evaluate the similarity between generated and ground-truth summaries and are widely used in code summarization.

2) Code Translation: Code translation aims to convert code written in one programming language to another [33]. This involves interpreting the logic and functionality of the original code and reproducing it accurately in the target language. We utilize the template described in [11] as the basic prompt:

"*Translate the above* [SOURCE] *code to* [TARGET]."

where [SOURCE] and [TARGET] specify the source and target programming languages for translation, respectively.

Datasets. The dataset for the code translation task is Avatar [34]. We use parallel functions in Java and Python to evaluate the task. The detailed data statistics are in shown Table I.

Metrics. To evaluate the code translation task, we adopt three additional metrics besides BLEU: Syntax Match (SM), Dataflow Match (DM), and CodeBLEU (CB) [35]. SM measures the proportion of matching subtrees in Abstract Syntax Trees (AST), while DM calculates the proportion of matching data flow edges. CB extends BLEU by incorporating syntactic and semantic features such as AST and data flow.

3) API Recommendation: API recommendation aims to suggest appropriate Application Programming Interface (API) calls for specific programming tasks. In this work, we focus on natural-language-based API recommendation, where the goal is to retrieve suitable APIs according to user' textual queries. We use the template described in [36] as the basic prompt.

"*Please recommend some suitable APIs for the given query.*"

Datasets. In API recommendation task, we adopt the widely used APIBENCH-Q [37] dataset. Following an 8:1:1 train/validation/test split, we randomly select 400 Python and 600 Java questions as the test set, with the rest queries grouped into the training and validation set. The detailed data statistics are shown in Table I.

Metrics. Following prior work [36], we utilize the widely used metrics in recommendation tasks to evaluate [37]–[39]: Success Rate (SR) and Mean Reciprocal Rank (MRR). SR@k evaluates a method's ability to suggest relevant APIs within the top-k recommendations, with k values of 1, 3, and 5 used for our evaluation. Meanwhile, MRR computes the average reciprocal rank of the first relevant API across all queries.

C. Large Code Models

To evaluate the effectiveness of APG methods, we utilize four open-source LCMs:

- **CodeLlama** [1] is a family of large language models for code based on Llama 2 [40] with state-of-the-art code generation, and blank infilling capabilities.
- **Qwen2.5-Coder** [3] is a LCM from Qwen2.5 series with 128K context length. It is trained on 5.5T tokens over source code and text-code alignment data, achieving strong performance in code generation and reasoning.
- **Deepseek-Coder** [2] is a series of LCMs trained on 2T tokens across over 80 programming languages. It achieves state-of-the-art performance among open-source code LLMs.
- **Magicoder** [41] is an instruction-tuned LCM, which first trains Deepseek-Coder on 75K OSS-INSTRUCT data and further trains the model on 110K Evol-Instruct data.

D. Implementation Details

In our study, We employ the following four open-source LCMs: Deepseek-Coder (Deepseek-Coder-6.7B-Instruct), Magicoder (Magicoder-S-DS-6.7B), CodeLlama (CodeLlama-13b-Instruct) and Qwen2.5-Coder (Qwen2.5-Coder-14B-Instruct). We download these models from HuggingFace[1] and deploy them locally. For the implementation of instruction generation (APE and OPRO) and multi-step reasoning methods (CoT, AutoCoT and Self-Plan), we directly use the replication packages released by the authors and adapt them to our tasks. As for the hyperparameters for the generation of LCMs, we adopt nuclear sampling with a top-p value of 0.95 and a temperature value of 0.2. The maximum generation token length is set to 256 for code translation, 512 for code summarization, and 128 for API recommendation.

All the experiments are conducted on an Ubuntu-20.04 server equipped with 4 * Nvidia A100 GPUs and each one has 40GB graphics memory. The source code is publicly accessible at https://anonymous.4open.science/r/Towards-Prompt-is-All-You-Need-5D66.

IV. Experimental Results

A. RQ1: Impact of automated instruction generation

Experimental Design. To answer this research question, we evaluate APE and OPRO against basic prompts (Section III-B) on API recommendation, code translation, and summarization tasks. All experiments are repeated five times, with the best performing instructions selected for the final evaluation.

[1]https://huggingface.co/models

TABLE II: Experiment results of instruction generation methods in code intelligence tasks.

Model	Approach	API Recommendation (NL-PL)				Code Translation (PL-PL)				Code Summarization (PL-NL)		
		Python				Python → Java				Python		
		SR@1	SR@3	SR@5	MRR	CB	SM	DM	BLEU	BLEU	ROUGE-L	METEOR
Deepseek-Coder	Basic	12.75	15.50	17.00	14.38	51.43	74.29	50.79	39.45	15.68	18.82	7.06
	APE	**15.25**	**17.75**	**18.75**	**16.56**	**65.36**	**76.05**	**68.00**	**58.92**	**15.87**	**20.60**	**8.66**
	OPRO	13.50	18.00	18.75	15.69	60.19	75.42	60.86	51.44	15.85	20.30	8.24
Magicoder	Basic	11.75	14.25	16.00	13.22	52.51	68.85	33.10	48.58	14.77	18.56	7.09
	APE	**14.50**	**17.00**	**19.75**	**16.23**	**63.02**	**76.57**	**68.35**	**52.98**	**15.83**	**20.44**	**8.54**
	OPRO	14.25	16.00	17.75	15.36	60.51	76.04	61.35	51.75	15.78	19.75	7.73
CodeLlama	Basic	10.50	13.75	14.75	12.00	56.29	70.05	57.48	48.20	9.24	10.78	4.75
	APE	**17.50**	**19.50**	**22.00**	**18.99**	**62.94**	**74.15**	**65.08**	**56.27**	**15.82**	**20.44**	**7.52**
	OPRO	15.25	17.25	18.25	16.39	58.60	72.24	61.61	50.00	15.82	19.96	6.56
Qwen2.5-Coder	Basic	10.25	13.25	14.50	12.31	56.18	70.29	57.52	48.73	12.47	14.36	5.82
	APE	**18.75**	**20.50**	**21.75**	**19.12**	**64.95**	**75.16**	**66.91**	**56.48**	**15.91**	**20.18**	**8.63**
	OPRO	15.50	17.25	18.50	16.08	59.44	72.68	63.84	50.36	14.76	18.68	7.58
Model	**Approach**	**API Recommendation (NL-PL)**				**Code Translation (PL-PL)**				**Code Summarization (PL-NL)**		
		Java				Java → Python				Java		
		SR@1	SR@3	SR@5	MRR	CB	SM	DM	BLEU	BLEU	ROUGE-L	METEOR
Deepseek-Coder	Basic	19.67	22.83	24.33	21.39	49.88	59.39	37.11	50.75	14.15	15.57	6.12
	APE	**21.67**	**26.50**	**28.00**	**24.22**	**58.61**	**61.98**	**41.94**	**64.92**	**15.02**	18.60	**7.24**
	OPRO	20.17	23.50	24.17	21.76	55.96	60.65	40.78	60.83	15.02	**18.69**	5.41
Magicoder	Basic	14.00	18.33	19.33	16.07	48.85	59.34	39.11	47.55	11.11	13.28	4.93
	APE	**17.83**	**22.33**	**23.50**	**20.08**	**54.52**	**63.01**	**42.75**	**52.64**	**14.65**	**17.83**	6.30
	OPRO	15.17	19.50	21.00	17.38	52.18	62.82	42.58	50.88	13.81	16.95	**6.48**
CodeLlama	Basic	18.67	24.00	26.50	21.63	50.57	59.02	38.55	51.54	4.71	5.37	2.41
	APE	**28.83**	**33.67**	**36.00**	**31.51**	**59.05**	56.57	**53.58**	**59.88**	**15.01**	**18.49**	**7.08**
	OPRO	25.17	29.33	31.67	27.54	55.46	**63.43**	43.11	50.53	15.01	18.46	6.91
Qwen2.5-Coder	Basic	19.83	24.17	26.33	21.29	51.23	59.67	39.41	52.36	10.64	11.18	6.53
	APE	**29.33**	**33.67**	**35.67**	**34.22**	**60.74**	**64.38**	**55.13**	**62.87**	**15.13**	**18.62**	**9.27**
	OPRO	26.50	30.17	31.83	27.18	56.52	61.33	45.77	54.09	14.92	17.14	8.03

TABLE III: Average instruction tokens for API recommendation, code translation and code summarization tasks.

Method	API Recommendation	Code Translation	Code Summarization
OPRO	41.92	36.78	68.54
APE	35.16 (**-6.76**)	34.27 (**-2.51**)	56.16 (**-12.38**)

(a) Instruction generated by APE

Please take the provided snippets of code written in Python and convert them into their equivalent Java code. Make sure to use **appropriate syntax, conventions, and structure** specific to Java programming language. Also, ensure that **the functionality of the original Python code is maintained** in the translated Java code.

(b) Instruction generated by OPRO

Translate the Python code to Java code. Your task is to write a program that takes the Python code as input and produces the Java code as output. The Python code was provided as input, and the Java code was produced as output. **The translation should be done correctly.** The friend **translated the Java code to Python code with no errors** and the Python code works as expected.

Fig. 3: An example of APE and OPRO generated instructions for the code translation task using Deepseek-Coder.

Analysis. Table II shows the results of instruction generation methods. Observations are as follows:

Automatic instruction generation methods consistently outperform basic prompts across all tasks and models. Both APE and OPRO demonstrate substantial improvements over basic prompts. In code summarization, APE and OPRO achieve average BLEU improvements of 50.80% and 48.40% respectively across all models and languages. For API recommendation, APE shows consistent gains across all metrics (41.56% for SR@1, 30.98% for SR@3, 30.11% for SR@5, and 37.01% for MRR), while OPRO also exhibits improvements (25.37% for SR@1, 17.55% for SR@3, 15.12% for SR@5, and 19.67% for MRR), though with smaller margins than APE. We also conduct t-test between APE/OPRO and the basic prompt among all tasks, and the results confirm statistically significant improvements ($p < 0.05$).

APE achieves better performance than OPRO despite using fewer instruction tokens, demonstrating its effectiveness and token efficiency. As shown in Table III, under the same experimental setup, APE consistently generates more concise instructions across all tasks: API recommendation (35.16 vs. 41.92 tokens), code translation (34.27 vs. 36.78 tokens), and code summarization (56.16 vs. 68.54 tokens), while maintaining better task performance.

To better understand this advantage, we compared the selection strategies and instruction content of APE and OPRO. Specifically, APE ranks candidate instructions using log prob-

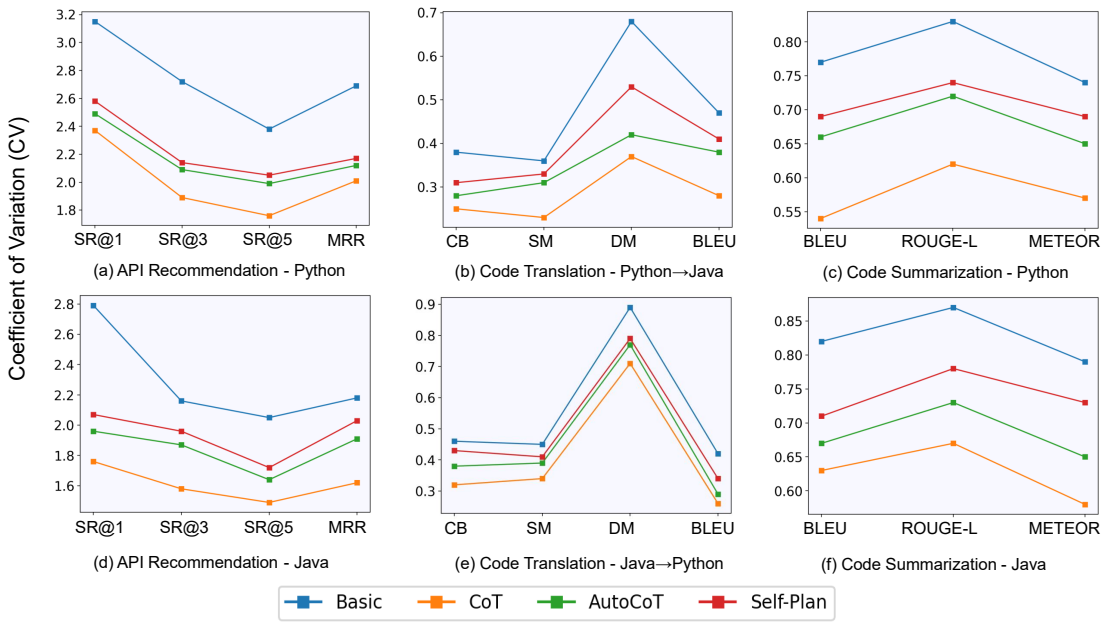

Fig. 4: Results of multi-step reasoning on three code intelligence tasks. The vertical axis means the average CV of each metric.

ability scores from the target model, whereas OPRO relies on scores generated directly by the LCMs. This difference in scoring may influence the alignment between the selected instruction and the underlying model's behavior. For example, as illustrated in Figure 3, although both instructions specify the source and target programming languages and emphasize translation quality, OPRO often includes redundant or less focused content, while APE produces more task-specific instructions that emphasize critical aspects such as appropriate syntax, naming conventions and structural consistency. This precise and concise guidance enables the LCM to perform better with fewer tokens.

Finding 1: Automatic instruction generation enhances LCMs in code intelligence tasks. APE surpasses OPRO in performance despite utilizing fewer tokens, demonstrating both effectiveness and token efficiency.

B. RQ2: Effectiveness of multi-step reasoning methods

Experimental Design. To investigate this RQ, we select three widely-used methods, including CoT, AutoCoT and Self-Plan. Templates and details of these methods are provided in our GitHub repository. To further assess robustness, we repeat each experiment five times and report the average performance. Additionally, we compute the Coefficient of Variation (CV = $\frac{\sigma}{\mu}$) across runs. A lower CV indicates more stable performance and stronger robustness.

Analysis. Results are shown in Table IV, and the corresponding CV values are illustrated in Figure 4 for method stability comparison. We can get the following insights through the results:

Multi-step reasoning methods can improve the performance of LCMs, with task-specific and detailed intermediate reasoning steps yielding the most notable gains. For example, in both code translation directions, CoT achieves improvements over the basic prompt of 21.70% in CB, 6.49% in SM, 27.96% in DM, and 34.34% in BLEU. In comparison, AutoCoT shows corresponding improvements of 15.70%, 4.24%, 24.03%, and 20.74%, while Self-Plan yields more modest gains of 6.86%, 2.38%, 4.90%, and 7.78%, respectively. Overall, CoT achieves the highest performance in almost all metrics (84/88) across all tasks. Statistical significance is confirmed by t-test ($p < 0.05$), demonstrating its superiority over other multi-step reasoning approaches. These results suggest that providing detailed intermediate steps (as in CoT) is more effective than decomposing the task into structured subtasks (as in Self-Plan) or relying on representative examples (as in AutoCoT), particularly in code intelligence tasks requiring task-specific understanding and logical reasoning.

CoT produces results that are more stable and less sensitive to code intelligence tasks. As shown in Figure 4, CoT consistently achieves the lowest CV values across all evaluation metrics and programming languages. In code summarization, CoT exhibits an average CV of 0.60, outperforming AutoCoT (0.68), Self-Plan (0.72), and basic prompts (0.80). For code translation, CoT demonstrates superior stability with BLEU score CVs of 0.26 (Java-to-Python) and 0.28 (Python-to-Java), compared to AutoCoT (0.29 and 0.38) and Self-Plan (0.34 and 0.41). In API recommendation, CoT achieves the lowest average CV values across all metrics: 2.07 for SR@1, 1.74 for SR@3, 1.63 for SR@5, and 1.82 for MRR. The consistent lowest CV across all three tasks highlights that CoT delivers more stable performance through

TABLE IV: Experiment results of multi-step reasoning methods in code intelligence tasks.

Model	Approach	API Recommendation (NL-PL)				Code Translation (PL-PL)				Code Summarization (PL-NL)		
		Python				Python → Java				Python		
		SR@1	SR@3	SR@5	MRR	CB	SM	DM	BLEU	BLEU	ROUGE-L	METEOR
Deepseek-Coder	Basic	12.75	15.50	17.00	14.38	51.43	74.29	50.79	39.45	15.68	18.82	7.06
	CoT	**14.75**	**19.25**	**21.00**	**17.05**	**66.22**	**76.65**	**69.19**	**59.83**	**15.89**	**20.57**	**9.16**
	AutoCoT	14.25	18.50	19.00	16.28	58.09	75.03	68.19	44.16	15.73	19.45	6.11
	Self-Plan	13.75	17.00	19.00	15.70	55.45	72.29	52.96	43.12	15.53	19.01	8.94
Magicoder	Basic	11.75	14.25	16.00	13.22	52.51	68.85	33.10	48.58	14.77	18.56	7.09
	CoT	**13.75**	**16.50**	**18.50**	**15.52**	**66.55**	75.48	**69.42**	**60.90**	**15.79**	**19.27**	**8.72**
	AutoCoT	12.50	15.75	17.50	14.37	65.68	**76.02**	68.65	58.93	15.71	19.14	8.09
	Self-Plan	10.75	15.25	17.25	13.34	56.41	73.13	34.75	50.46	15.76	19.26	8.38
CodeLlama	Basic	10.50	13.75	14.75	12.00	56.29	70.05	57.48	48.20	9.24	10.78	4.75
	CoT	**22.75**	**30.75**	**33.00**	**26.67**	**61.71**	73.78	**66.41**	**53.10**	**15.90**	**20.32**	**8.81**
	AutoCoT	21.75	26.75	29.00	24.36	61.20	**73.90**	64.83	52.68	15.87	20.21	5.60
	Self-Plan	19.50	25.00	26.75	22.30	58.43	70.83	58.06	50.10	13.40	16.20	7.24
Qwen2.5-Coder	Basic	10.25	13.25	14.50	12.31	56.18	70.29	57.52	48.73	12.47	14.36	5.82
	CoT	**22.25**	**31.50**	**32.75**	**28.52**	**61.92**	**73.93**	**66.53**	**58.69**	**16.21**	**21.33**	**8.84**
	AutoCoT	21.50	26.25	28.25	26.73	61.47	72.85	64.87	55.91	15.87	20.26	5.53
	Self-Plan	19.25	25.50	26.25	22.18	58.46	71.17	58.26	49.78	13.32	16.14	7.23
Model	**Approach**	**API Recommendation (NL-PL)**				**Code Translation (PL-PL)**				**Code Summarization (PL-NL)**		
		Java				Java → Python				Java		
		SR@1	SR@3	SR@5	MRR	CB	SM	DM	BLEU	BLEU	ROUGE-L	METEOR
Deepseek-Coder	Basic	19.67	22.83	24.33	21.39	49.88	59.39	37.11	50.75	14.15	15.57	6.12
	CoT	**25.00**	**28.33**	**29.83**	**26.73**	**61.22**	**62.81**	**42.38**	**69.60**	**15.07**	**18.65**	**7.69**
	AutoCoT	23.17	27.00	28.50	25.25	56.15	61.94	40.44	56.51	14.94	18.00	5.30
	Self-Plan	22.17	25.33	26.67	23.84	51.71	60.77	39.77	52.63	14.96	17.65	7.50
Magicoder	Basic	14.00	18.33	19.33	16.07	48.85	59.34	39.11	47.55	11.11	13.28	4.93
	CoT	**20.50**	**25.67**	**28.00**	**23.31**	**61.13**	**62.22**	**40.88**	**70.48**	**14.96**	17.64	**7.60**
	AutoCoT	20.33	23.83	26.17	22.40	57.76	61.44	40.42	64.26	14.94	**17.83**	5.25
	Self-Plan	15.50	21.33	24.67	18.91	55.95	60.01	40.46	61.08	14.93	17.57	7.39
CodeLlama	Basic	18.67	24.00	26.50	21.63	50.57	59.02	38.55	51.54	4.71	5.37	2.41
	CoT	**24.83**	**32.50**	34.50	**28.66**	**63.06**	**63.51**	**43.22**	**72.60**	**15.07**	**18.63**	**7.51**
	AutoCoT	24.17	31.83	**35.00**	28.31	60.08	60.83	40.30	67.97	15.03	18.44	6.48
	Self-Plan	23.17	26.17	27.50	24.80	53.75	62.53	42.73	53.17	13.57	15.96	6.83
Qwen2.5-Coder	Basic	19.83	24.17	26.33	21.29	51.23	59.67	39.41	52.36	10.64	11.18	6.53
	CoT	**24.50**	**33.33**	**34.67**	**29.58**	**64.41**	**65.87**	**45.58**	**73.76**	**15.12**	**19.63**	**7.95**
	AutoCoT	23.33	30.83	34.17	28.25	61.38	61.14	42.62	68.23	15.03	18.44	6.48
	Self-Plan	22.33	26.17	27.50	26.52	54.97	61.91	41.84	56.35	13.54	15.92	6.82

multi-step reasoning, enabling it to mitigate task performance inconsistency caused by prompt variations.

Finding 2: Multi-step reasoning methods effectively enhance the performance of LCMs in code intelligence tasks, while CoT demonstrates more consistent gains and lower sensitivity to task variation compared to other reasoning methods.

C. RQ3: Can we further improve the performance of current APG methods?

Experimental Design. While previous results indicate the individual effectiveness of IG and MSR methods, it remains unclear whether their combination can yield further improvements. Therefore, in RQ3, we investigate whether combining APE (for IG) and CoT (for MSR) into a unified APE-CoT framework can achieve better performance than individual method. As shown in Figure 5, the APE-CoT selects the top-ranked instruction through iterative APE generation and then incorporates CoT reasoning steps to construct the final prompt. Due to time constraints, we sample 10% of the code summarization test set (1,095 Java and 1,491 Python examples) while using full datasets for code translation and API recommendation given their smaller sizes.

 # Instruction Generated by APE
Translate Java code to equivalent Python code. Each input-output pair shows the Java code and its corresponding Python code that performs the same function. Make sure to use appropriate syntax, conventions, and structure specific to Java programming language.

Here are reasoning steps of the given code:
{$ Given Java Code}

 # Multi-step Reasoning Generated by CoT
Let's think step by step:
Step 1. Use a for loop to iterate through …
Step 2. For each card , generate a random …
Step 3. Swap positions of the current card …
…
Therefore, the translated Python Code is:

Fig. 5: An example of APE-CoT prompt in code translation.

Analysis. The evaluation results are presented in Table V,

TABLE V: Experiment results of the combined methods in code intelligence tasks.

Model	Approach	API Recommendation (NL-PL)				Code Translation (PL-PL)				Code Summarization (PL-NL)		
		Python				Python → Java				Python		
		SR@1	SR@3	SR@5	MRR	CB	SM	DM	BLEU	BLEU	ROUGE-L	METEOR
Deepseek-Coder	APE	15.25	17.75	18.75	16.56	65.36	76.05	68.00	58.92	15.64	19.09	9.29
	CoT	14.75	19.25	21.00	17.05	66.22	76.65	69.19	**59.83**	16.04	**19.67**	9.47
	APE-CoT	**17.25**	**21.00**	**21.75**	**19.16**	**70.78**	**88.47**	**75.71**	58.77	**16.06**	19.60	**9.50**
Magicoder	APE	14.25	17.00	19.75	16.23	63.02	**76.57**	68.35	52.98	15.99	19.51	9.18
	CoT	13.75	16.50	18.50	15.52	66.55	75.48	69.42	60.90	15.57	19.02	8.56
	APE-CoT	**14.50**	**18.75**	**20.00**	**16.61**	**75.46**	76.45	**69.71**	**77.13**	**16.04**	**19.61**	**8.97**
CodeLlama	APE	17.50	19.50	22.00	18.99	62.94	74.15	65.08	56.27	13.03	15.72	7.83
	CoT	22.75	30.75	33.00	26.67	61.71	73.78	66.41	53.10	16.00	19.60	9.00
	APE-CoT	**29.00**	**32.75**	**34.75**	**31.18**	**64.53**	**74.57**	**66.44**	**58.50**	**16.06**	**19.65**	**9.58**
Qwen2.5-Coder	APE	18.75	20.50	21.75	19.12	64.95	75.16	66.91	56.48	15.93	19.41	8.69
	CoT	22.25	31.50	32.75	28.52	61.92	73.93	66.53	58.69	15.99	19.36	8.74
	APE-CoT	**31.75**	**34.25**	**37.25**	**31.62**	**69.02**	**76.37**	**69.81**	**61.02**	**17.24**	**20.52**	**9.35**
Model	**Approach**	**API Recommendation (NL-PL)**				**Code Translation (PL-PL)**				**Code Summarization (PL-NL)**		
		Java				Java → Python				Java		
		SR@1	SR@3	SR@5	MRR	CB	SM	DM	BLEU	BLEU	ROUGE-L	METEOR
Deepseek-Coder	APE	21.67	26.50	28.00	24.22	58.61	61.98	41.94	64.92	15.01	17.85	7.58
	CoT	25.00	28.33	29.83	26.73	61.22	62.81	42.38	69.60	14.96	17.94	7.80
	APE-CoT	**26.17**	**29.17**	**30.17**	**27.67**	**62.13**	**63.46**	**43.52**	**70.54**	**15.09**	**18.15**	**8.01**
Magicoder	APE	17.83	22.33	23.50	20.08	54.52	63.01	**42.75**	52.64	14.63	17.31	7.41
	CoT	20.50	25.67	**28.00**	23.31	61.13	62.22	40.88	70.48	15.03	17.85	7.53
	APE-CoT	**21.50**	**26.17**	27.67	**23.84**	**62.69**	**63.92**	42.63	**71.89**	**15.07**	**18.16**	**7.84**
CodeLlama	APE	28.83	33.67	36.00	31.51	59.05	56.57	**53.58**	59.88	13.37	15.96	7.13
	CoT	24.83	32.50	34.50	28.66	63.06	63.51	43.22	72.60	15.03	17.86	7.62
	APE-CoT	**32.17**	**40.83**	**43.17**	**36.76**	**63.44**	**64.00**	42.33	**73.57**	**15.09**	**18.12**	**8.00**
Qwen2.5-Coder	APE	29.33	33.67	35.67	34.22	60.74	64.38	55.13	62.87	14.96	17.88	7.58
	CoT	24.50	33.33	34.67	29.58	64.41	65.87	45.58	73.76	15.02	17.81	7.49
	APE-CoT	**34.67**	**42.83**	**45.17**	**38.46**	**66.54**	**67.61**	**56.47**	**74.59**	**15.11**	**18.14**	**8.97**

and we summarize the main findings as follows:

Combining generated instruction and reasoning steps improves the performance of APG methods across code intelligence tasks. According to Table V, APE-CoT consistently outperforms both individual APE and CoT methods across all evaluated models and tasks. The combined approach achieves optimal performance in 93.18% (41/44) of Python task metrics and 95.45% (42/44) of Java task metrics, suggesting the complementary benefits of both components. For instance, in Python API recommendation, APE-CoT with Qwen2.5-Coder achieves 31.75% SR@1, substantially outperforming individual APE (18.75%) and CoT (22.25%) methods. Similarly, in Python to Java translation, APE-CoT consistently delivers the highest performance across different models, such as achieving 75.46% CodeBLEU with Magicoder compared to 63.02% for APE and 66.55% for CoT. These results demonstrate that combining APG techniques enhances code intelligence performance, with t-tests confirming that APE-CoT significantly outperforms each individual method ($p < 0.05$).

APE-CoT exhibits different improvements across code intelligence tasks, with particularly notable gains in NL-PL scenarios. The combined approach achieves the most substantial improvements in NL-PL tasks (API recommendation), with an average improvement of 22.35% over individual APE and CoT methods across all models and metrics. For PL-PL tasks (code translation), APE-CoT shows moderate gains with an average improvement of 8.75% compared to individual methods. In contrast, PL-NL tasks (code summarization) exhibit the modest improvements, with APE-CoT achieving an average gain of 7.28% over APE and CoT approaches. These results indicate that APE-CoT is particularly effective for tasks with programming language outputs, showing substantial improvements in both API recommendation and code translation , while exhibiting relatively smaller gains in code summarization that requires natural language generation.

> **Finding 3:** Combining instruction generation and multi-step reasoning further improves individual APG methods for code intelligence tasks, particularly excelling in code output tasks (NL-PL and PL-PL) while exhibiting limited gains in PL-NL scenario.

D. RQ4: Performance of APE-CoT in Industrial Scenario

Experimental Design. To evaluate the effectiveness of APE-CoT in real-world industrial settings, we conduct experiments on an internal dataset from Tencent. Specifically, we construct an API query dataset of 1,000 samples collected from the C++ codebase of WeChat, referred to as WeChat-Bench. We choose C++ for this evaluation as it is the primary language used in large-scale projects at Tencent, and thus the dataset better reflects the practical challenges encountered by developers in industrial scenarios. Given the scale and com-

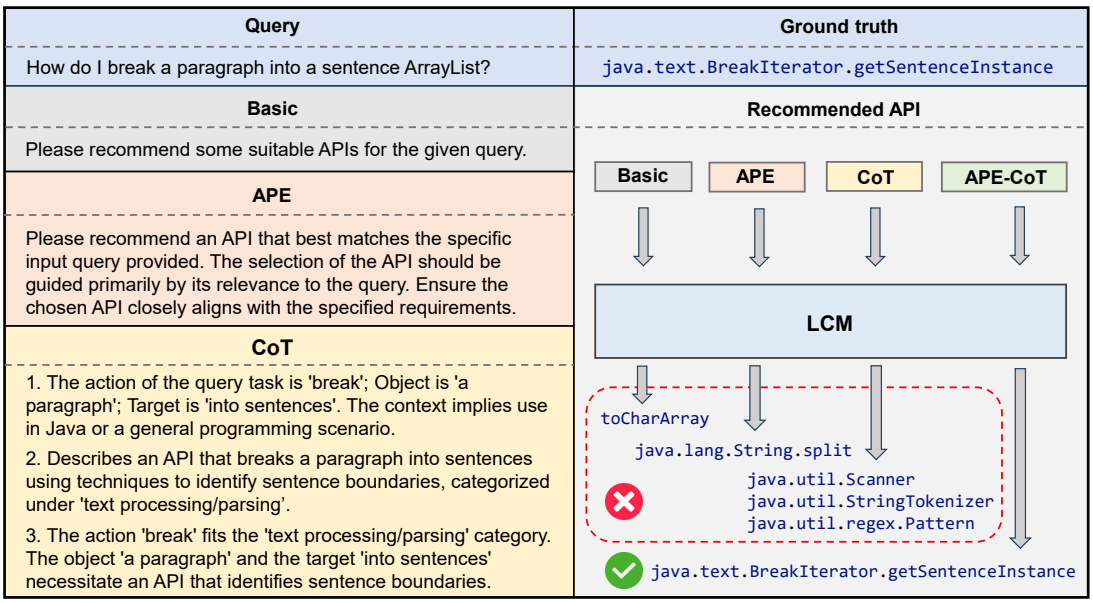

Fig. 6: A case study on the API recommendation task with four prompts. The LCM is Codellama.

TABLE VI: Experimental results on the WeChat-Bench dataset for API recommendation in the industry scenario.

Models	Approach	WeChat-Bench			
		SR@1	SR@3	SR@5	MRR
Deepseek-Coder	Basic	3.90	5.10	6.00	3.90
	APE-CoT	**7.80**	**9.80**	**10.30**	**7.70**
Magicoder	Basic	0.80	1.20	1.70	0.70
	APE-CoT	**4.00**	**5.90**	**6.70**	**4.00**
CodeLlama	Basic	12.80	14.00	14.80	12.80
	APE-CoT	**13.60**	**15.80**	**16.70**	**13.50**
Qwen2.5-Coder	Basic	10.80	12.00	12.60	9.90
	APE-CoT	**13.00**	**13.60**	**14.20**	**12.00**

plexity of WeChat, WeChat-Bench provides a representative and challenging production-level enterprise environment for validating the effectiveness of APE-CoT in industrial scenario.

Analysis. The experimental results in Table VI show that the APE-CoT framework consistently outperforms the basic prompt across all evaluated LCMs on the industrial WeChat-Bench dataset. For example, Deepseek-Coder achieves an SR@1 of 7.80% with APE-CoT, doubling the performance of the basic prompt (3.90%). Similar improvements are observed for Magicoder, CodeLlama, and Qwen2.5-Coder, with APE-CoT yielding higher scores in all metrics (SR@1, SR@3, SR@5, and MRR). Notably, although the metric scores on WeChat-Bench are lower than those on public benchmarks, this is likely due to the increased difficulty and domain specificity of internal industrial data. Despite these challenging conditions, APE-CoT still surpasses the basic prompt, demonstrating both its effectiveness and generalizability for API recommendation in the real-world industrial scenario.

Finding 4: APE-CoT consistently outperforms the basic prompt for API recommendation, indicating effectiveness and generalizability in industrial scenario.

V. Discussion

A. Why does APE-CoT work?

The effectiveness of APE-CoT lies in the combination of task-specific instruction and multi-step reasoning. The instruction provides task-specific guidance, and the multi-step reasoning enables the model to enhance the understanding of task. Together, they guide the LCM to generate more accurate and contextually aligned code intelligence task results. The following sections illustrate these two aspects in detail through an API recommendation example.

1) Instruction Generation for Task-Specific Guidance: When prompted with only the basic query, the LCM recommends the API "*toCharArray*", which is driven by a narrow focus on keywords like "Array", leading to a less relevant recommendation. In contrast, the APE method introduces task-specific instructions, offering the model clearer guidance and helping it better understand the user's intent. As a result, the model suggests "*java.lang.String.split*", a much more appropriate API for string segmentation. This highlights how instruction generation can guide the model toward more contextually aligned recommendations.

2) Multi-Step Reasoning for Enhanced understanding of Task Instructions: While multi-step reasoning can guide the LCM through a sequence of logical steps, it may still fall short when task instructions are underspecified. In the case, the CoT prompt guides the LCM to generate APIs such as "*java.util.Scanner, java.util.StringTokenizer,*

java.util.regex.Pattern". These recommendations suggest an attempt to segment a paragraph by first identifying sentence boundaries through tokenization and pattern matching. While this reflects a certain level of structured reasoning, the model still fails to capture the core intent of sentence-level splitting, resulting in suboptimal API choices. By integrating multi-step reasoning with task-specific instructions, the APE-CoT method enables the model to better understand the instruction and progressively refine its decisions. As a result, the LCM identifies the correct API: "*java.text.BreakIterator.getSentenceInstance*", demonstrating the combination of instruction and multi-step reasoning leads to more contextually appropriate results.

B. Implications of Findings

This study conducts an empirical evaluation of automated prompt generation methods in code intelligence tasks. In this section, we discuss the implications of our study from the perspective of researchers and developers.

Researchers: Our study indicates that APG methods can create effective prompts that improve LCMs performance in code intelligence tasks. Our findings provide potential research directions for researchers in the code intelligence community:

- The integration of APG methods shows superior performance, with the APE-CoT approach surpassing individual APG strategies. This suggests the potential of exploring alternative APG combinations for further improvements in code intelligence tasks.
- APG methods are more effective for code output tasks (NL-PL and PL-PL) but show limited gains in natural language generation (PL-NL). Future work should explore enhancing APG performance in PL-NL scenarios.

Developers: Automated crafting effective prompts is critical for developers working with LCMs. Based on our findings, we offer the following practical recommendations:

- Consider adopting automated prompt generation to reduce reliance on manual prompts and enhance both stability and effectiveness in code intelligence tasks.
- Combine instruction generation and reasoning methods to enhance LCMs performance, particularly for code output tasks such as API recommendation and code translation.

C. Threats to Validity

We identify the following threats to validity of our study:

The selection of evaluation datasets. The quality and scale of datasets can affect the results of the experiment. Although we choose the widely-used datasets, the diversity of the dataset is still limited. In future work, we will evaluate the effectiveness of APG methods on more datasets.

The selection of LCMs. Due to computational resource constraints, the maximum parameters of the chosen LCMs is 14B, which may not fully reflect the performance of APG methods on larger LCMs. To further validate the generalizability of our findings, we intend to extend our experiments to a broader range of LCMs.

The selection of evaluation metrics. The current evaluation focuses on accuracy-based and generation-quality metrics. While these provide a solid basis for comparison, they may not fully capture model robustness or efficiency. Future work will explore broader evaluation metrics for more comprehensive assessment of APG methods.

VI. Related work

A. Automatic Prompt Generation (APG)

APG has emerged as a promising direction to improve the adaptability and performance of LCMs [17], [42], [43], especially in tasks where manually crafting effective prompts is challenging. Recent studies have proposed various APG methods aimed at designing more effective prompts with minimal human effort. One aspect of studies focuses on generating clearer and more effective task descriptions. For instance, Automatic Prompt Engineering (APE) [17] automatically optimizes or selects prompts that better align with task objectives. Another aspect of work enhances the reasoning capability of models through structured prompting methods, such as Chain-of-Thought (CoT) prompting [23], which guide models through intermediate reasoning steps to improve final output quality. Our work builds upon these foundations by focusing on empirical strategies for automating prompt construction, with the goal of efficiently automatically generating prompts that maximize LCM performance on code intelligence tasks.

B. In-Context Learning (ICL)

In-Context Learning is an alternative training-free paradigm to improve LCMs performance on downstream tasks [12], [44]–[47]. Instead of relying on fine-tuning with the large-scale supervised dataset, ICL utilizes a small set of labeled samples from the training data to create task prompts. These prompts guide the model's responses during inference [48]. This method is typically classified by the number of examples provided, ranging from zero-shot (no examples) to few-shot learning (a limited number of examples). Recent studies in ICL have explored the impact of the order of examples on performance [49] and the enhancement of diversity among these examples for improved compositional generalization [50]. Moreover, there is a growing interest in applying ICL to code intelligence tasks [51]–[53], including techniques like using BM-25 retrieval for constructing relevant demonstrations [53]. In contrast, our work adapts automated prompting generation methods to enhance LCMs performance in code intelligence tasks, rather than repeatedly selecting exemplary examples to facilitate models' understanding of context and tasks.

VII. Conclusion

In this paper, we empirically investigate two important parts in automated prompting generation (APG) for code intelligence tasks, including Instruction Generation (IG) and Multi-Step Reasoning (MSR). For each part, we evaluate widely-used APG methods on four open-source LCMs and three popular code intelligence tasks. Our study indicates that the two parts in APG can greatly enhance the performance of code intelligence tasks compared to basic prompts. Based on these findings, we propose APE-CoT, a novel approach that

combines the most effective methods from each part. Experiments on public benchmarks and in the industrial scenario of WeChat demonstrate that our approach achieves non-trivial improvements and confirms its practical applicability.

ACKNOWLEDGMENT

This research is supported by the National Natural Science Foundation of China under project (No. 62472126, 62276075), Natural Science Foundation of Guangdong Province (Project No. 2023A1515011959), Shenzhen-Hong Kong Jointly Funded Project (Category A, No. SGDX20230116 091246007).

REFERENCES

[1] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. Canton-Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, and G. Synnaeve, "Code llama: Open foundation models for code," *CoRR*, vol. abs/2308.12950, 2023.

[2] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. K. Li, F. Luo, Y. Xiong, and W. Liang, "Deepseek-coder: When the large language model meets programming - the rise of code intelligence," *CoRR*, vol. abs/2401.14196, 2024.

[3] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu, K. Dang, A. Yang, R. Men, F. Huang, X. Ren, X. Ren, J. Zhou, and J. Lin, "Qwen2.5-coder technical report," *CoRR*, vol. abs/2409.12186, 2024.

[4] W. U. Ahmad, S. Chakraborty, B. Ray, and K. Chang, "A transformer-based approach for source code summarization," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020* (D. Jurafsky, J. Chai, N. Schluter, and J. R. Tetreault, eds.), pp. 4998–5007, Association for Computational Linguistics, 2020.

[5] J. A. Li, Y. Li, G. Li, X. Hu, X. Xia, and Z. Jin, "Editsum: A retrieve-and-edit framework for source code summarization," *CoRR*, vol. abs/2308.13775, 2023.

[6] Y. Kang, Z. Wang, H. Zhang, J. Chen, and H. You, "Apirecx: Cross-library API recommendation via pre-trained language model," in *EMNLP (1)*, pp. 3425–3436, Association for Computational Linguistics, 2021.

[7] Z. Li, C. Li, Z. Tang, W. Huang, J. Ge, B. Luo, V. Ng, T. Wang, Y. Hu, and X. Zhang, "Ptm-apirec: Leveraging pre-trained models of source code in API recommendation," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 3, pp. 72:1–72:30, 2024.

[8] B. Rozière, M. Lachaux, L. Chanussot, and G. Lample, "Unsupervised translation of programming languages," in *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual* (H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, eds.), 2020.

[9] Z. Yang, B. Hu, A. Han, S. Huang, and Q. Ju, "CSP: code-switching pre-training for neural machine translation," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP 2020, Online, November 16-20, 2020* (B. Webber, T. Cohn, Y. He, and Y. Liu, eds.), pp. 2624–2636, Association for Computational Linguistics, 2020.

[10] C. Wang, Y. Yang, C. Gao, Y. Peng, H. Zhang, and M. R. Lyu, "No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022* (A. Roychoudhury, C. Cadar, and M. Kim, eds.), pp. 382–394, ACM, 2022.

[11] R. Pan, A. R. Ibrahimzada, R. Krishna, D. Sankar, L. P. Wassi, M. Merler, B. Sobolev, R. Pavuluri, S. Sinha, and R. Jabbarvand, "Lost in translation: A study of bugs introduced by large language models while translating code," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*, pp. 82:1–82:13, ACM, 2024.

[12] M. Geng, S. Wang, D. Dong, H. Wang, G. Li, Z. Jin, X. Mao, and X. Liao, "Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*, pp. 39:1–39:13, ACM, 2024.

[13] W. Sun, C. Fang, Y. You, Y. Miao, Y. Liu, Y. Li, G. Deng, S. Huang, Y. Chen, Q. Zhang, H. Qian, Y. Liu, and Z. Chen, "Automatic code summarization via chatgpt: How far are we?," *CoRR*, vol. abs/2305.12865, 2023.

[14] J. White, S. Hays, Q. Fu, J. Spencer-Smith, and D. C. Schmidt, "Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design," *CoRR*, vol. abs/2303.07839, 2023.

[15] H. Tian, W. Lu, T. O. Li, X. Tang, S. Cheung, J. Klein, and T. F. Bissyandé, "Is chatgpt the ultimate programming assistant - how far is it?," *CoRR*, vol. abs/2304.11938, 2023.

[16] B. Paranjape, S. M. Lundberg, S. Singh, H. Hajishirzi, L. Zettlemoyer, and M. T. Ribeiro, "ART: automatic multi-step reasoning and tool-use for large language models," *CoRR*, vol. abs/2303.09014, 2023.

[17] Y. Zhou, A. I. Muresanu, Z. Han, K. Paster, S. Pitis, H. Chan, and J. Ba, "Large language models are human-level prompt engineers," in *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*, OpenReview.net, 2023.

[18] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. R. Narasimhan, and Y. Cao, "React: Synergizing reasoning and acting in language models," in *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*, OpenReview.net, 2023.

[19] J. Liu, A. Liu, X. Lu, S. Welleck, P. West, R. L. Bras, Y. Choi, and H. Hajishirzi, "Generated knowledge prompting for commonsense reasoning," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022* (S. Muresan, P. Nakov, and A. Villavicencio, eds.), pp. 3154–3169, Association for Computational Linguistics, 2022.

[20] T. Shin, Y. Razeghi, R. L. L. IV, E. Wallace, and S. Singh, "Autoprompt: Eliciting knowledge from language models with automatically generated prompts," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP 2020, Online, November 16-20, 2020* (B. Webber, T. Cohn, Y. He, and Y. Liu, eds.), pp. 4222–4235, Association for Computational Linguistics, 2020.

[21] C. Yang, X. Wang, Y. Lu, H. Liu, Q. V. Le, D. Zhou, and X. Chen, "Large language models as optimizers," *CoRR*, vol. abs/2309.03409, 2023.

[22] S. Gao, X. Wen, C. Gao, W. Wang, H. Zhang, and M. R. Lyu, "What makes good in-context demonstrations for code intelligence tasks with llms?," in *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*, pp. 761–773, IEEE, 2023.

[23] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. H. Chi, Q. V. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," in *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022* (S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, eds.), 2022.

[24] Z. Zhang, A. Zhang, M. Li, and A. Smola, "Automatic chain of thought prompting in large language models," in *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*, OpenReview.net, 2023.

[25] X. Jiang, Y. Dong, L. Wang, Q. Shang, and G. Li, "Self-planning code generation with large language model," *CoRR*, vol. abs/2303.06689, 2023.

[26] Z. Li, C. Wang, P. Ma, C. Liu, S. Wang, D. Wu, C. Gao, and Y. Liu, "On extracting specialized code abilities from large language models: A feasibility study," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*, pp. 74:1–74:13, ACM, 2024.

[27] S. Gao, C. Gao, Y. He, J. Zeng, L. Nie, X. Xia, and M. R. Lyu, "Code structure-guided transformer for source code summarization," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 1, pp. 23:1–23:32, 2023.

[28] M. Geng, S. Wang, D. Dong, H. Wang, G. Li, Z. Jin, X. Mao, and X. Liao, "An empirical study on using large language models for multi-intent comment generation," *arXiv preprint arXiv:2304.11384*, 2023.

[29] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *CoRR*, vol. abs/2102.04664, 2021.

[30] K. Papineni, S. Roukos, T. Ward, and W. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, July 6-12, 2002, Philadelphia, PA, USA*, pp. 311–318, ACL, 2002.

[31] C.-Y. Lin, "ROUGE: A package for automatic evaluation of summaries," in *Text Summarization Branches Out*, (Barcelona, Spain), pp. 74–81, Association for Computational Linguistics, July 2004.

[32] S. Banerjee and A. Lavie, "METEOR: an automatic metric for MT evaluation with improved correlation with human judgments," in *Proceedings of the Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization@ACL 2005, Ann Arbor, Michigan, USA, June 29, 2005*, pp. 65–72, Association for Computational Linguistics, 2005.

[33] J. D. Weisz, M. J. Muller, S. Houde, J. T. Richards, S. I. Ross, F. Martinez, M. Agarwal, and K. Talamadupula, "Perfection not required? human-ai partnerships in code translation," in *IUI '21: 26th International Conference on Intelligent User Interfaces, College Station, TX, USA, April 13-17, 2021* (T. Hammond, K. Verbert, D. Parra, B. P. Knijnenburg, J. O'Donovan, and P. Teale, eds.), pp. 402–412, ACM, 2021.

[34] W. U. Ahmad, M. G. R. Tushar, S. Chakraborty, and K. Chang, "AVATAR: A parallel corpus for java-python program translation," in *Findings of the Association for Computational Linguistics: ACL 2023, Toronto, Canada, July 9-14, 2023* (A. Rogers, J. L. Boyd-Graber, and N. Okazaki, eds.), pp. 2268–2281, Association for Computational Linguistics, 2023.

[35] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, "Codebleu: a method for automatic evaluation of code synthesis," *CoRR*, vol. abs/2009.10297, 2020.

[36] Y. Chen, C. Gao, M. Zhu, Q. Liao, Y. Wang, and G. Xu, "Apigen: Generative API method recommendation," in *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2024, Rovaniemi, Finland, March 12-15, 2024*, pp. 171–182, IEEE, 2024.

[37] Y. Peng, S. Li, W. Gu, Y. Li, W. Wang, C. Gao, and M. R. Lyu, "Revisiting, benchmarking and exploring API recommendation: How far are we?," *IEEE Trans. Software Eng.*, vol. 49, no. 4, pp. 1876–1897, 2023.

[38] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep API learning," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016* (T. Zimmermann, J. Cleland-Huang, and Z. Su, eds.), pp. 631–642, ACM, 2016.

[39] M. Wei, N. S. Harzevili, Y. Huang, J. Wang, and S. Wang, "CLEAR: contrastive learning for API recommendation," in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pp. 376–387, ACM, 2022.

[40] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. Canton-Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. Kloumann, A. Korenev, P. S. Koura, M. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. E. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, and T. Scialom, "Llama 2: Open foundation and fine-tuned chat models," *CoRR*, vol. abs/2307.09288, 2023.

[41] Y. Wei, Z. Wang, J. Liu, Y. Ding, and L. Zhang, "Magicoder: Source code is all you need," *CoRR*, vol. abs/2312.02120, 2023.

[42] Q. Guo, R. Wang, J. Guo, B. Li, K. Song, X. Tan, G. Liu, J. Bian, and Y. Yang, "Connecting large language models with evolutionary algorithms yields powerful prompt optimizers," in *ICLR*, OpenReview.net, 2024.

[43] R. Pryzant, D. Iter, J. Li, Y. T. Lee, C. Zhu, and M. Zeng, "Automatic prompt optimization with "gradient descent" and beam search," in *EMNLP*, pp. 7957–7968, Association for Computational Linguistics, 2023.

[44] A. Ni, S. Iyer, D. Radev, V. Stoyanov, W. Yih, S. I. Wang, and X. V. Lin, "LEVER: learning to verify language-to-code generation with execution," in *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA* (A. Krause, E. Brunskill, K. Cho, B. Engelhardt, S. Sabato, and J. Scarlett, eds.), vol. 202 of *Proceedings of Machine Learning Research*, pp. 26106–26128, PMLR, 2023.

[45] A. Patel, B. Li, M. S. Rasooli, N. Constant, C. Raffel, and C. Callison-Burch, "Bidirectional language models are also few-shot learners," in *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*, OpenReview.net, 2023.

[46] J. Y. Khan and G. Uddin, "Automatic code documentation generation using GPT-3," in *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*, pp. 174:1–174:6, ACM, 2022.

[47] Y. Wei, C. S. Xia, and L. Zhang, "Copiloting the copilots: Fusing large language models with completion engines for automated program repair," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023* (S. Chandra, K. Blincoe, and P. Tonella, eds.), pp. 172–184, ACM, 2023.

[48] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual* (H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, eds.), 2020.

[49] Y. Lu, M. Bartolo, A. Moore, S. Riedel, and P. Stenetorp, "Fantastically ordered prompts and where to find them: Overcoming few-shot prompt order sensitivity," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022* (S. Muresan, P. Nakov, and A. Villavicencio, eds.), pp. 8086–8098, Association for Computational Linguistics, 2022.

[50] I. Levy, B. Bogin, and J. Berant, "Diverse demonstrations improve in-context compositional generalization," in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023* (A. Rogers, J. L. Boyd-Graber, and N. Okazaki, eds.), pp. 1401–1422, Association for Computational Linguistics, 2023.

[51] T. Ahmed and P. T. Devanbu, "Few-shot training llms for project-specific code-summarization," in *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*, pp. 177:1–177:5, ACM, 2022.

[52] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pp. 1482–1494, IEEE, 2023.

[53] N. Nashid, M. Sintaha, and A. Mesbah, "Retrieval-based prompt selection for code-related few-shot learning," in *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pp. 2450–2462, IEEE, 2023.