ClarifyCoder: Instructing Code Large Language Models to Ask Clarifying Questions

JIE JW WU, University of British Columbia, Canada
MANAV CHAUDHARY, International Institute of Information Technology, India
FATEMEH H. FARD, University of British Columbia, Canada

Large language models (LLMs) have significantly improved their ability to perform tasks in the field of code generation. However, there is still a gap between LLMs being capable coders and being top-tier software engineers. Based on the observation that top-level software engineers often ask clarifying questions to reduce *Ambiguity* in both requirements and coding solutions, we argue that the same should be applied to LLMs for code generation tasks. Although the most recent trend is using LLM-based agents to iterate the code generation process, we argue that LLMs themselves should have the ability to ask clarifying questions when needed. Our study is motivated by the observation that the instruction tuning methods in Code LLMs enforce the models to generate code, regardless of any issues in the problem description. As a result, the state-of-the-art LLMs still often generate code when the problem description as input has issues. In this research, we propose ClarifyCoder to address this gap. Our goal is to develop a new instruction-tuning approach to empower the model to have the ability to ask clarifying questions when needed for coding tasks. Firstly, we develop a novel data synthesis method to generate problem descriptions where clarification questions are needed using the existing programming datasets. Then, we construct the clarification-aware data for fine-tuning. Finally, we fine-tune the model to ask clarifying questions when the model does not fully understand the problem description, instead of generating code directly. Our experimental results show the effectiveness of ClarifyCoder in communication ability.

ACM Reference Format:
Jie JW Wu, Manav Chaudhary, and Fatemeh H. Fard. 2025. ClarifyCoder: Instructing Code Large Language Models to Ask Clarifying Questions. 1, 1 (January 2025), 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Large language models (LLMs) [5, 19–21], such as OpenAI's Codex [4], AlphaCode [9], and CodeGen [15], possess a significantly capable ability to generate code snippets from natural language requirements. However, there have been several reported issues in LLMs, including problems with intent specification, problem decomposition [18], code quality, hallucination, communication [23] and overconfidence [11, 12], as well as usability [10]. The research gap needs to be addressed to enable the transition from task-driven AI copilot to goal-driven AI pair programmers [7]. Our study applies a communication lens to address this gap and provides a solution for it.

On the communication aspect, there is still a gap between LLMs being capable coders and being top-tier software engineers [23]. Given a software engineering task in real-world enterprises, software engineers use various ways of communication, such as asking more questions in 1:1 conversations, group meetings, and Slack channels to obtain

Authors' addresses: Jie JW Wu, University of British Columbia, 3333 University Way, Kelowna, B.C., V1V 1V7, Canada, jie.jw.wu@ubc.ca; Manav Chaudhary, International Institute of Information Technology, Professor CR Road, Hyderabad, 500032, India, manav.chaudhary@reseach.iiit.ac.in; Fatemeh H. Fard, University of British Columbia, 3333 University Way, Kelowna, B.C., V1V 1V7, Canada, fatemeh.fard@ubc.ca.

Manuscript submitted to ACM

more information and reduce *Ambiguity* about the detailed requirements, the context of the projects, and the design alternatives. Proactive and effective communication is a critical skill in practice for professional software engineers to accomplish their software engineering tasks reliably with high quality [8, 13, 14, 16, 22]. However, in the recent study [23] to benchmark the communication skills of Code LLMs, it was shown that more than 60% of responses from Code LLMs still generate code rather than ask questions when the problem descriptions are manually modified according to different clarification categories. Although the most recent trend is using LLM-based agents to iterate the code generation process, still, we argue that LLMs themselves should have the ability to ask clarifying questions when needed, which is a general capability that can be applied to other software engineering tasks.

Our study is motivated by the observation that the instruction tuning methods in Code LLMs force the models to generate code, regardless of any issues in the problem description. This issue is similar to the hallucination issue of LLMs, and there are several works [24] to address the hallucination problem in the Natural Language Processing (NLP) domain. If a model is trained on generating code for problem descriptions, the model has a natural tendency to generate code even when there appears to be issues or ambiguity in the problem descriptions. On the other hand, if the model has not been trained on asking clarifying questions to clarify on the ambiguities that lie in the problem descriptions, the model would not learn to ask effective clarifying questions for code generation tasks. Therefore, the main idea of our study is to fine-tune the Code LLMs to get the capability in asking effective clarifying questions that are needed to complete the software engineering tasks. We aim to answer the following research questions:

- **RQ1: How effective are the data synthesis methods in data construction for fine-tuning?**
- **RQ2: To what extent is the model effective in its ability in asking clarifying questions with the Clarify-Aware Instruction Tuning?**

In this research, we propose a new instruction-tuning approach, ClarifyCoder, to empower the model to have the ability to ask effective clarifying questions when needed for coding tasks. Firstly, we develop a novel data synthesis method to generate problem descriptions where clarification questions are needed. Then we construct the clarification-aware data for fine-tuning by leveraging the data synthesis method on the existing code generation datasets. Finally, we fine-tune the model to ask clarifying questions when the model does not fully understand the problem description, instead of generating code directly. Our experimental results show the effectiveness of ClarifyCoder in communication ability.

To summarize, our contributions are listed as follows:

- We are the first to investigate the ability of Code LLMs to ask clarifying questions for code generation tasks at the model level.
- We propose novel data synthesis methods to automatically modify the problem description so that the description needs clarifying questions. The data synthesis methods are mainly used for the data construction for model fine-tuning. They are potentially also useful in finding blind spots of both the model and evaluation metrics for Code LLMs.
- We present a novel fine-tuning method, **Clarify-Aware Instruction Tuning**, which first constructs the dataset using the proposed data synthesis methods, then performs fine-tuning to enable the model to ask clarifying questions needed for completing the coding tasks.

2 CLARIFY-AWARE DATA CONSTRUCTION

This section details the methodology used to generate modified coding problem descriptions and their corresponding clarifying questions. Our objective was to produce problem statements characterized by ambiguity, inconsistency, or incompleteness, and their corresponding clarifying questions, thereby creating a robust dataset for training ClarifyCoder.

2.1 Dataset

We utilize the APPS dataset, which comprises of 10,000 coding problems sourced from diverse open-access platforms like Codeforces and Kattis. Each problem is accompanied by multiple test cases and human-written solutions, covering a range of complexities from introductory to collegiate competition levels, with an average description length of 293.2 words.

2.2 Model

Given the constraints of our budget, we opted to employ Google Gemini due to its strong performance and accessibility via a free research API. This API facilitates smooth integration into our scripts, enabling efficient model calls and robust error management.

2.3 Generating Modified Problems

We instruction-tune the Gemini model to modify original problem statements through a chain-of-thought knowledge-infused prompting strategy. This includes modifications specific to each modification category. Here are the instructions we used to make the model generate modified problems from their original counterparts:

- **Ambiguous Modification**: Based on the knowledge that ambiguous problem descriptions can be created by introducing multiple valid interpretations or unspecified details, think step-by-step to rewrite the given coding problem description and make it ambiguous.
- **Incomplete Modification**: Based on the knowledge that removing some of the key concepts and conditions that are crucial for solving the problem makes it incomplete, think step-by-step to rewrite the given coding problem description and make it incomplete.
- **Inconsistent Modification**: Based on the knowledge that a problem becomes inconsistent if some statements in the description show conflict, think step-by-step to rewrite the given coding problem description and make it inconsistent.

2.4 Generating Clarifying Questions

After generating the modified problem descriptions, we proceed to create the corresponding clarifying questions. For each modified problem, we input both the modified and the original problem descriptions into the model using the following instructions:

- **Ambiguous**: You are given a coding problem description and an ambiguous version of the same problem. Your task is to assess the ambiguous problem description, identify specific points of ambiguity, and ask necessary clarifying questions to resolve the ambiguity and reach the clarity present in the original problem. Please note that a problem statement is ambiguous if it includes multiple valid interpretations or has unspecified details.
- **Incomplete**: You are given a coding problem description and an incomplete version of the same problem. Your task is to assess the incomplete problem description, identify specific points of incompleteness, and ask

necessary clarifying questions to resolve the incompleteness and reach the clarity present in the original problem. Please note that absence of some of the key concepts and conditions that are crucial for solving the problem makes it incomplete.

- **Inconsistent**: You are given a coding problem description and an inconsistent version of the same problem. Your task is to assess the inconsistent problem description, identify specific points of inconsistency, and ask necessary clarifying questions to resolve the inconsistency and reach the clarity present in the original problem. Please note that a problem description becomes inconsistent if some statements in the description show conflict.

We ensure that the generated clarifying questions must focus solely on the modified problem descriptions, without referencing the original versions, since the original version would never be available to the finetuned model. We provide the original questions in the instruction so that the generated clarifying questions are relevant to the problem at hand, i.e., generating code, ensuring that the model does not ask irrelevant questions. We use the following instruction at the end of our prompt to achieve this:

- Ensure that each question targets a specific point to achieve clarity similar to the original problem. When generating these questions, do not reference or mention the original problem description in any way. Frame the clarifying questions as if you have only seen the modified problem description, without acknowledging the existence of the original version.

2.5 Data Consolidation and Error Management

To ensure the integrity of the data generation process, various error handling mechanisms are implemented. These include tracking empty outputs and internal server errors, allowing for retries while preserving progress. Additionally, a checkpoint system is integrated, enabling the pipeline to resume from the last successful operation in case of interruptions.

In the final step, we consolidate the modified problems and their corresponding clarifying questions into a unified dataset tailored for fine-tuning the language model. This dataset contains three primary fields: problem, answer, and type. The types of problem-answer pairs are categorized as follows:

- Type: Original: The original problem statements from the APPS dataset, paired with human-written solutions. (Note: this has been removed)
- Type: Ambiguous: Problem statements modified by Gemini to introduce ambiguity, with clarifying questions as the "answer."
- Type: Inconsistent: Problem statements modified to become inconsistent, with corresponding clarifying questions.
- Type: Incomplete: Problem statements modified to be incomplete, with related clarifying questions.

Our structured approach ensures the creation of high-quality training data that enhances the capability of the model to engage in effective clarification during code generation tasks.

3 CLARIFY-AWARE INSTRUCTION TUNING

Given the clarify-aware data being constructed and available, we have several option to perform fine-tuning.

3.1 Clarify-Aware Tuning

We use the standard fine-tuning process for the LLMs, given the constructed data. We will use the standard cross-entropy loss as the objective of fine-tuning, and calculate the loss of answers. Specifically, for each sample (q, a) in the dataset D, the standard tuning of language models is to generate a with q as input, following the standard cross-entropy loss L:

$$L = -\sum_{t=1}^{|o|} P(o_t|o_{<t}, q) \tag{1}$$

3.2 Combining Standard and Clarify-Aware Tuning

3.3 Evaluation Metrics

Following the evaluation of HumanEvalComm [23], in our study, if the models will output either code or clarifying questions, we evaluate the Pass@1, Test Pass Rates, Communication Rate and Good Question Rates for the models. This ensures we evaluate the quality of the clarifying questions and how do they improve the pass rates.

On the other hand, if the model outputs a confidence score on whether to ask clarify questions or generate code, similar to the R-Tuning evaluation protocol, we can use Average Precision (AP) based on the Test Pass Rate for a problem description.

4 EXPERIMENTAL EVALUATION

4.1 Experimental Setup

4.2 Models

We plan to perform our evaluation on five widely used LLMs. This includes three open-sourced instruction-tuned Code LLMs, one open-sourced instruction-tuned LLM, and one commercial LLM. For open-source models, we plan to use models with the largest possible model size within our limited computing resources in our evaluation.

- **CodeLlama** (Instruction tuned version, 13B) [17] is an open-source LLM released by Meta for coding, built on top of Llama 2, with foundation models and instruction models. CodeLlama was chosen because of its wide usage and top performance in HumanEval. We tested the instruction model CodeLlama-Instruct-13B in our experiment since we did not have the computing resources to run models with 34B. The same applies to the rest open-source models.
- **DeepSeek Coder** (Instruction tuned version, 7B) [6] is an open-source Code LLM trained on both 87% code and 13% natural language. Each of the models was pre-trained on 2 trillion tokens. We selected this model because it achieved top 5 performance in Big Code Models Leaderboard [1] on the HuggingFace platform. The Big Code Models Leaderboard [1] evaluates the performance of base multilingual code generation models on the HumanEval benchmark and MultiPL-E. We used the model of 7 billion parameters in the evaluation.
- **DeepSeek Chat** (Instruction tuned version, 7B) [3] is an open-source LLM released by DeepSeek AI, trained on datasets of 2 trillion tokens. We selected this model because we wanted to evaluate the communication skills of models trained from different sources such as natural languages, code, and a combination of both. We compared its performance with the DeepSeek Coder to understand whether more natural languages in pre-training are beneficial to communication skills. We used the model of 7 billion parameters in the evaluation.
- **CodeQwen1.5 Chat** (Instruction tuned version, 7B) [2] is an open-souce Code LLM released by Qwen Team, trained on 3 trillion tokens of code data. CodeQwen1.5 Chat is the Code-Specific version of Qwen1.5. The model

is a transformer-based decoder-only language model and includes group query attention (GQA) for efficient inference. We selected this model because it achieved top 5 performance in Big Code Models Leaderboard [1].

- **ChatGPT**, released by OpenAI are powerful models for generation tasks. We used parameter-frozen versions of models (gpt-3.5-turbo-0125) to ensure the reproducibility of the evaluation results.

Note that all of the evaluated models above are instruction-tuned models because, in the evaluation, the ability to ask clarifying questions with the given prompts is needed for the models. Besides instruction-tuned models, there are also foundation models, but we didn't report results for foundation models. We found that foundation models without instruction tuned are not suitable for our evaluation, because their task is only to complete code and are not capable of instructions such as "either generate code or ask clarifying questions".

4.3 RQ1 Results

4.4 RQ2 Results

REFERENCES

[1] Hugging Face Accessed 2024. Big Code Models Leaderboard. Hugging Face. https://huggingface.co/spaces/bigcode/bigcode-models-leaderboard Accessed on April 29, 2024.

[2] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. 2023. Qwen technical report. arXiv preprint arXiv:2309.16609 (2023).

[3] Xiao Bi, Deli Chen, Guanting Chen, Shanhuang Chen, Damai Dai, Chengqi Deng, Honghui Ding, Kai Dong, Qiushi Du, Zhe Fu, et al. 2024. Deepseek llm: Scaling open-source language models with longtermism. arXiv preprint arXiv:2401.02954 (2024).

[4] 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 (2021).

[5] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, and D. Jiang et al. 2020. CodeBERT: A Pre-trained Model for Programming and Natural Languages. arXiv preprint arXiv:2002.08155 (2020).

[6] 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 (2024).

[7] Ahmed E Hassan, Gustavo A Oliva, Dayi Lin, Boyuan Chen, Zhen Ming, et al. 2024. Rethinking Software Engineering in the Foundation Model Era: From Task-Driven AI Copilots to Goal-Driven AI Pair Programmers. arXiv preprint arXiv:2404.10225 (2024).

[8] Mehdi Jazayeri. 2004. The education of a software engineer. In Proceedings. 19th International Conference on Automated Software Engineering, 2004. IEEE, xviii–xxvii.

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

[10] Jenny T Liang, Chenyang Yang, and Brad A Myers. 2023. Understanding the Usability of AI Programming Assistants. arXiv preprint arXiv:2303.17125 (2023).

[11] Yue Liu, Thanh Le-Cong, Ratnadira Widyasari, Chakkrit Tantithamthavorn, Li Li, Xuan-Bach D Le, and David Lo. 2023. Refining ChatGPT-Generated Code: Characterizing and Mitigating Code Quality Issues. arXiv preprint arXiv:2307.12596 (2023).

[12] Zhijie Liu, Yutian Tang, Xiapu Luo, Yuming Zhou, and Liang Feng Zhang. 2023. No Need to Lift a Finger Anymore? Assessing the Quality of Code Generation by ChatGPT. arXiv preprint arXiv:2308.04838 (2023).

[13] Ian R McChesney and Seamus Gallagher. 2004. Communication and co-ordination practices in software engineering projects. Information and Software Technology 46, 7 (2004), 473–489.

[14] Ivan Mistrík, John Grundy, Andre Van der Hoek, and Jim Whitehead. 2010. Collaborative software engineering: challenges and prospects. Springer.

[15] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. arXiv preprint arXiv:2203.13474 (2022).

[16] Roger S Pressman. 2005. Software engineering: a practitioner's approach. Palgrave macmillan.

[17] 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 (2023).

[18] Advait Sarkar, Andrew D Gordon, Carina Negreanu, Christian Poelitz, Sruti Srinivasa Ragavan, and Ben Zorn. 2022. What is it like to program with artificial intelligence? arXiv preprint arXiv:2208.06213 (2022).

[19] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan. 2020. Intellicode Compose: Code Generation Using Transformer. In Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 1433–1443.

[20] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, ÅĄ. Kaiser, and I. Polosukhin. 2017. Attention is All You Need. In Advances in Neural Information Processing Systems, Vol. 30.

[21] Y. Wang, W. Wang, S. Joty, and S. C. Hoi. 2021. CodeT5: Identifier-Aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. arXiv preprint arXiv:2109.00859 (2021).

[22] Jim Whitehead. 2007. Collaboration in software engineering: A roadmap. In Future of Software Engineering (FOSE'07). IEEE, 214–225.

[23] Jie JW Wu and Fatemeh H Fard. 2024. Benchmarking the Communication Competence of Code Generation for LLMs and LLM Agent. arXiv preprint arXiv:2406.00215 (2024).

[24] Hanning Zhang, Shizhe Diao, Yong Lin, Yi Fung, Qing Lian, Xingyao Wang, Yangyi Chen, Heng Ji, and Tong Zhang. 2024. R-Tuning: Instructing Large Language Models to Say âĂŸI DonâĂŹt KnowâĂŹ. In Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers). 7106–7132.