# CLARIGEN: Bridging Instruction Gaps via Interactive Clarification in Code Generation

**Chunyu Miao**[1], **Yibo Wang**[1], **Langzhou He**[1], **Liancheng Fang** [1], **Philip S. Yu** [1]
[1]University of Illinois at Chicago
{cmiao8, ywang633, lhe24, psyu}@uic.edu

**Abstract**

Large Language Models(LLMs) excel at generating code but often struggle when faced with incomplete or underspecified instructions. Drawing on the practice of experienced developers who seek clarification before coding, we introduce a framework that integrates a clarifying Q&A phase into the code generation process. Instead of working blindly from vague prompts, our approach encourages users to refine their requirements, enabling the LLM to produce more contextually informed and accurate code. We apply this technique to a range of challenging tasks, demonstrating that high-quality clarifications substantially improve both code correctness and reliability. Our results highlight a promising avenue for enhancing human-LLM collaboration, making generated code solutions more aligned with user intent and reducing the need for subsequent revisions.

## Introduction

In recent years, Large Language Models (LLMs) have demonstrated remarkable capabilities in generating code (Jiang et al. 2024a). Given instructions for a coding task, these models can produce code snippets that closely align with the specified requirements. To measure their effectiveness across different dimensions, researchers have introduced a wide range of benchmarks (Hendrycks et al. 2021; Chen et al. 2021; Du et al. 2023; Zhuo et al. 2024), each designed to evaluate various aspects of code generation quality.

As LLMs have advanced, however, many of these earlier benchmarks have lost their discriminatory power, with state-of-the-art models now easily surpassing them. As a result, the focus has shifted toward more complex and realistic scenarios (Du et al. 2023; Farchi et al. 2024; Jimenez et al. 2024). Under these conditions, a critical challenge arises: the initial instructions given to the model are often incomplete or underspecified. While an LLM may produce semantically correct code according to the provided instructions, the resulting output may fail to pass comprehensive test suites due to missing contextual details.

This issue is not limited to academic settings. In real-world software development, engineers frequently receive only high-level requirements, which may lack the granularity needed to implement a solution that meets all specifications. When critical information is absent, the code produced by an LLM—or even a human programmer—can struggle to fulfill strict testing criteria, ultimately hindering reliable deployment.

A common approach that skilled programmers use in these scenarios is to seek clarification before coding. For example, when faced with vague or incomplete requirements, a developer might ask follow-up questions to ensure full understanding before proceeding with implementation. Inspired by this practice, our work proposes a novel code generation framework, ClariGen (depicted in Figure 1), that simulates this behavior. Instead of directly producing code from an underspecified prompt, the LLMs first generate clarifying questions, prompting the user to provide essential details. With these enriched requirements in hand, the model can then generate code that aligns more closely with the user's goals, thereby increasing the likelihood of passing strict validation tests. In this way, our approach improves both the quality and reliability of code generation by encouraging a more interactive and context-aware development process.

In our empirical evaluation, we demonstrate that incorporating clarification questions into the code generation process yields measurable improvements across multiple dimensions. Compared to baseline models that generate code directly from the initial prompt, our approach shows higher Pass@1 on comprehensive test suites and exhibits greater robustness to vague or underspecified instructions. Furthermore, we analyze the effectiveness of using clarification questions generated by different models, revealing that higher-quality Q&A content consistently leads to stronger performance gains. Although some tasks still regress from success to failure, the overall trends suggest that augmenting the coding instructions through an interactive clarification process substantially enhances both the accuracy and reliability of the generated code.

## Related Work

### Code Generation with LLMs

Code generation using LLMs has garnered significant attention, particularly because these models, trained on exten-

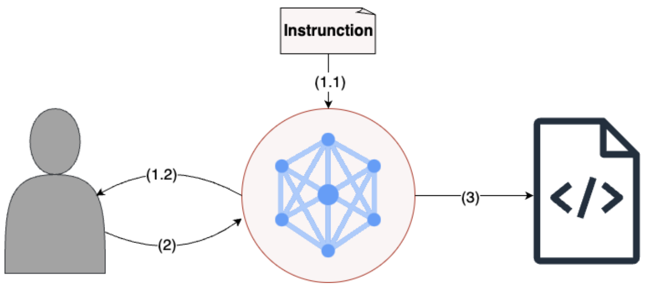

Figure 1: An overview of the ClariGen framework. (1.1) The model receives an initial instruction. (1.2) The model queries the user for clarification on missing or ambiguous details. (2) The user provides additional information in response. (3) Incorporating these clarifications, the model produces the final code output.

sive code corpora, exhibit emergent capabilities in generating high-quality code (Li et al. 2022; Austin et al. 2021). Current research predominantly focuses on enhancing code quality through various strategies, such as pre-processing or post-processing of instructions and code (Jiang et al. 2024b; Olausson et al. 2024). However, as evaluation tasks become more complex, the instructions in benchmarks are generally insufficient to guide LLMs in generating code that passes strict tests. ClarifyGPT (Mu et al. 2024) addresses this challenge by detecting ambiguities in the generated code and subsequently providing corrections to achieve the correct implementation. Although ClarifyGPT demonstrates significant improvements, it remains unclear whether LLMs can directly ask questions based on given instructions to further enhance code quality. In our work, we directly generate questions from the provided instructions and use the answers, informed by software requirements, to simulate the software development process and improve code quality.

### Asking Clarification Questions

Asking Clarification Questions (ACQ) helps prevent misunderstandings in human-human and human-machine communication by eliciting missing information and resolving ambiguities (Shi, Feng, and Lipani 2022; Zou et al. 2023). Widely used in NLP tasks like dialogue systems and query refinement (Zamani et al. 2020; Rahmani et al. 2023), ACQ remains underexplored in code generation. Recognizing that developers often seek clarification before coding, we extend ACQ to this domain. By prompting for additional details to clarify vague or incomplete requirements, we enable LLMs to produce more accurate, context-aware code. This iterative approach enhances reliability and success rates in challenging tasks, aligning code generation more closely with real-world software development practices.

## Methodology

Our proposed framework, illustrated in Figure 1, is designed to simulate the real-world workflow of a programmer tackling an underspecified coding task with partial software requirements. It decomposes the code generation process into three key stages: (1) Clarification Question Generation, (2) Question Answering, and (3) Code Generation.

### Clarification Question Generation

Given an initial coding task instruction $\psi$, the Large Language Model $M_l$ emulates a developer confronted with ambiguous requirements. In this step, $M_l$ identifies missing details or unclear specifications—often related to the program's intended function, inputs, and outputs—and generates a set of clarifying questions. Formally:

$$\{q_i\}_{i=1}^{n_q} = M_l(\psi).$$

The number of questions $n_q$ emerges dynamically, reflecting the complexity and incompleteness of the given instruction.

### Answering Questions

Once the clarification questions $\{q_i\}_{i=1}^{n_q}$ are generated, they are presented to the user $U$. Drawing on their domain expertise and the underlying requirements, the user responds with a set of corresponding answers $\{a_i\}_{i=1}^{n_q}$. This interactive exchange refines the previously underspecified instruction $\psi$, enriching it with critical contextual details and ensuring that the specification aligns more closely with the intended functionality.

### Code Generation

With the original instruction $\psi$ now supplemented by the clarified requirements $\{q_i, a_i\}_{i=1}^{n_q}$, the LLM proceeds to generate the final code:

$$c = M_l(\psi, \{q_i\}_{i=1}^{n_q}, \{a_i\}_{i=1}^{n_q}).$$

Through this iterative clarification cycle, the produced code $c$ becomes more context-aware, accurate, and closely aligned with the user's intended goals.

### Evaluation

Most existing code generation benchmarks provide only problem statements and expected solutions, but do not supply additional clarifications or answers to potential follow-up questions. To integrate our framework seamlessly with these datasets, we simulate the user's role by employing an LLM as an answer provider, as shown in Figure 2. This simulation ensures that our methodology can be directly applied to existing benchmarks without manual intervention, maintaining consistency and allowing for automated evaluation of Q&A-enhanced code generation.

## Experiment and Result

In this section, we present our experimental evaluation, aiming to answer the following research questions:

1. **Framework Effectiveness:** Will our proposed framework yield higher-quality code if additional clarifications are provided?
2. **Model Generality:** How do different LLMs perform when integrated into our framework?
3. **Cross-Model Consistency:** Can the generated QA pairs be effectively transferred to other LLMs to improve code generation quality?

4. **Performance Variance:** What is the performance differential between the model and the baseline?

The following subsections detail our experimental setup and results, offering insights into each of these questions.

## Experimental Setup

Our experiments are conducted on the BigCodeBench-Hard subset (Zhuo et al. 2024), which comprises approximately 150 challenging, user-facing tasks selected from the BigCodeBench dataset. To evaluate performance, we employ the widely used Pass@K metric, which measures the effectiveness of LLMs in generating correct code by determining the probability that at least one of the top-K generated samples passes all test cases. In this experiment, we set K to 1, aligning with the most practical application of LLMs in code generation, where typically only a single solution is sought. This setting closely mirrors real-world usage scenarios, emphasizing the model's ability to produce correct and functional code on the first attempt. We evaluate four LLMs in our framework: GPT-4 (OpenAI et al. 2024), Gemini (Team et al. 2024), Qwen2.5-32B-Coder (Hui et al. 2024), and Qwen2.5-72B-Instruct (Yang et al. 2024).

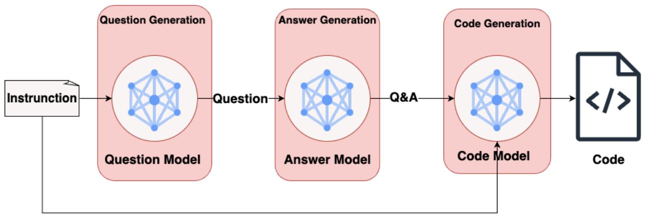

Figure 2: Illustration of the evaluation framework. The instruction is first processed by a Question Model to generate clarification questions. These questions are then answered by the Answer Model, and the resulting Q&A pairs are used by the Code Model to produce the final code output.

The experimental procedure is divided into three main stages, as depicted in Figure 2: (1) Question Generation, (2) Answer Generation, and (3) Code Generation. See Appendix A for a detailed look at how the prompts for different tasks.

Question Generation: In this stage, the selected question LLM is supplied with the initial coding task instruction and tasked with generating a set of clarification questions. This process mirrors how a developer would seek additional details from stakeholders when requirements are incomplete. By pinpointing ambiguities and gaps, the LLM produces questions that target crucial missing information, setting the stage for more precise and context-aware code generation.

Answer Generation: Once the clarification questions are formulated, we use GPT-4 to produce corresponding answers. For this purpose, GPT-4 has access to the original instruction, the canonical code (as a reference), and the generated questions. The answers are crafted to focus strictly on the software requirements, intentionally excluding any direct implementation details. This approach simulates real-world development scenarios, where engineers rely on specifications and high-level requirements rather than detailed code insights before beginning implementation. By adhering to these guidelines, the answers become both practical and representative of typical software engineering workflows.

| Model | Pass@1(%) | Improve |
|---|---|---|
| GPT-4 w/o Q&A | 34.67 | - |
| GPT-4 w/ Q&A | 35.33 | 0.67 |
| Gemini w/o Q&A | 26.67 | - |
| Gemini w Q&A | 31.33 | 4.67 |
| Qwen2.5-32B-Coder w/o Q&A | 33.33 | - |
| Qwen2.5-32B-Coder w Q&A | 36.67 | 3.33 |
| Qwen2.5-72B-Instruct w Q&A | 30.00 | - |
| Qwen2.5-72B-Instruct w Q&A | 33.33 | 3.33 |

Table 1: Pass@1 of different LLMs on the code-generation task, comparing their performance with and without an interactive Q&A stage.

Code Generation: With the finalized clarification questions and their corresponding answers in hand, we then prompt each LLM to produce the final code solution. The input to the model includes the original instruction as well as the question-answer pairs, enabling the model to generate code that better aligns with the clarified requirements.

## Framework Effectiveness

To assess the effectiveness of our framework, we conduct experiments in which the same model is used both to generate clarification questions and to produce the final code. Table 1 compares our interactive approach to a baseline where code is generated directly from the original instruction.

As shown in the table, incorporating the question-answer phase significantly improves the quality of the generated code. By enriching the original instructions with additional details gleaned from answers, our framework enables the model to better understand the underlying requirements. These findings suggest that our approach is particularly valuable in practical scenarios where initial instructions are incomplete or vague, ultimately enhancing code generation reliability and overall utility.

## Model Generality

Although all models show improvements over the baseline, their relative gains vary. Notably, GPT-4, which achieves state-of-the-art performance without clarification, is overtaken by Qwen2.5-32B-Coder once question-answer pairs are introduced. This indicates that even models starting from a lower baseline can outperform stronger models if given the opportunity to clarify ambiguities in the instructions.

Gemini, despite having the weakest baseline performance, experiences the largest relative increase in Pass@1 when clarification is provided. This finding suggests that models initially struggling with incomplete instructions may benefit disproportionately from the added context.

Both Qwen2.5-32B-Coder and Qwen2.5-72B-Instruct show substantial improvements in Pass@1. Among all models tested, Qwen2.5-32B-Coder ultimately achieves the highest Pass@1 once clarification is introduced, demonstrat-

| Model | Pass@1(%) | Improve |
|---|---|---|
| GPT-4 w/o Q&A | 34.67 | - |
| GPT-4 w/ Q&A from itself | 35.33 | 0.67 |
| GPT-4 w/ Q&A from Gemini | 35.33 | 0.67 |
| GPT-4 w/ Q&A from Qwen | 38.00 | 3.33 |
| Gemini w/o Q&A | 26.67 | - |
| Gemini w/ Q&A from itself | 31.33 | 4.67 |
| Gemini w/ Q&A from GPT-4 | 29.33 | 2.67 |
| Gemini w/ Q&A from Qwen | 31.33 | 4.67 |
| Qwen w/o Q&A | 33.33 | - |
| Qwen w/ Q&A from itself | 36.67 | 3.33 |
| Qwen w/ Q&A from GPT-4 | 38.00 | 4.67 |
| Qwen w/ Q&A from Gemini | 31.33 | -2.00 |

Table 2: Cross-model Q&A evaluation results. Pass@1 of three LLMs (GPT-4, Gemini, and Qwen) are reported under four conditions: without Q&A, with self-generated Q&A, and with Q&A provided by different models.

ing the strong potential of combining effective question-asking with high-quality model architectures.

In summary, these results highlight that the effectiveness of our framework is not restricted to a single model. Instead, it generalizes well, allowing various LLMs—regardless of their baseline capabilities—to generate code more successfully when provided with additional clarification.

## Cross-Model Consistency

To further investigate the effectiveness of our framework, we examined scenarios where the model generating clarification questions differs from the model ultimately producing the code. We focus on three representative LLMs: GPT-4, Gemini, and Qwen2.5-32B-Coder (hereafter Qwen). These models were selected based on their performance profiles in Table 1: GPT-4, with its large number of parameters, consistently achieves top-tier baseline results; Gemini, on the other hand, demonstrates the weakest coding ability; and Qwen, despite having fewer parameters, shows substantial gains when integrated with our approach. This range of capabilities makes them ideal candidates for evaluating the cross-model consistency of Q&A-driven clarifications. The results of these experiments are summarized in Table 2.

These results underscore the significance of Q&A quality for enhancing code generation. Across models, incorporating clarifications generally improves performance, even when the Q&A content is not ideal. For example, Gemini's lower-quality Q&A still slightly benefits GPT-4, demonstrating GPT-4's resilience. In contrast, Qwen's Q&A consistently delivers strong gains for both GPT-4 and Gemini, suggesting that better questions uncover critical details and lead to more accurate solutions. When Qwen relies on Gemini's Q&A, however, performance declines, indicating that poor-quality clarifications can misdirect the model. Overall, these findings highlight that while most Q&A interactions help, high-quality clarifications—like those from Qwen—are especially effective in improving reliability and accuracy.

## Performance Variance

In addition to improving accuracy, the introduction of Q&A pairs can also introduce variability into the results. Specifically, for some coding tasks, incorporating Q&A leads to a decrease in performance for certain cases, as certain clarification pairs may unintentionally misguide the model, causing it to produce incorrect solutions. This phenomenon is reflected in Table 3, where we report cases in which tasks that were previously solved successfully by the baseline now fail, as well as instances where incorporating Q&A results in additional test suite failures. As shown in Table 3, the degree

| Model | # Pass | # More Failure |
|---|---|---|
| GPT-4 w/ Q&A from itself | 14 | 25 |
| GPT-4 w/ Q&A from Gemini | 18 | 29 |
| GPT-4 w/ Q&A from Qwen | 11 | 17 |
| Gemini w/ Q&A from itself | 13 | 37 |
| Gemini w/ Q&A from GPT-4 | 8 | 18 |
| Gemini w/ Q&A from Qwen | 9 | 23 |
| Qwen w/ Q&A from itself | 12 | 18 |
| Qwen w/ Q&A from GPT-4 | 8 | 16 |
| Qwen w/ Q&A from Gemini | 20 | 40 |

Table 3: Variance analysis comparing our method to the baseline. '# Pass' counts tasks that were previously successful but now fail due to Q&A. '# More Failure' indicates additional test suites that fail.

of negative impact correlates with the quality of the Q&A pairs. Higher-quality Q&A from Qwen and GPT-4 tend to cause fewer misleading scenarios, resulting in fewer regression cases compared to self-generated or Gemini-derived Q&A. In contrast, lower-quality Q&A, particularly from Gemini, significantly increases the likelihood of misinterpretations and performance drops. These findings highlight that while Q&A can enrich the instruction context, its effectiveness and stability critically depend on the quality of the clarification content itself.

## Conclusion

We introduce ClariGen, a framework that integrates clarifying questions into the code generation process. By prompting users to refine underspecified instructions before generating code, ClariGen significantly improves Pass@1 results and yields more reliable solutions, even when working with models that previously struggled with vague prompts. Moreover, our cross-model experiments show that high-quality clarification pairs are transferable and beneficial across diverse LLMs. Overall, the approach moves code generation closer to realistic software development practices, laying a foundation for more robust, context-aware AI coding assistants.

## References

Austin, J.; Odena, A.; Nye, M.; Bosma, M.; Michalewski, H.; Dohan, D.; Jiang, E.; Cai, C.; Terry, M.; Le, Q.; and Sutton, C. 2021. Program Synthesis with Large Language Models. arXiv:2108.07732.

Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; de Oliveira Pinto, H. P.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; Ray, A.; Puri, R.; Krueger, G.; Petrov, M.; Khlaaf, H.; Sastry, G.; Mishkin, P.; Chan, B.; Gray, S.; Ryder, N.; Pavlov, M.; Power, A.; Kaiser, L.; Bavarian, M.; Winter, C.; Tillet, P.; Such, F. P.; Cummings, D.; Plappert, M.; Chantzis, F.; Barnes, E.; Herbert-Voss, A.; Guss, W. H.; Nichol, A.; Paino, A.; Tezak, N.; Tang, J.; Babuschkin, I.; Balaji, S.; Jain, S.; Saunders, W.; Hesse, C.; Carr, A. N.; Leike, J.; Achiam, J.; Misra, V.; Morikawa, E.; Radford, A.; Knight, M.; Brundage, M.; Murati, M.; Mayer, K.; Welinder, P.; McGrew, B.; Amodei, D.; McCandlish, S.; Sutskever, I.; and Zaremba, W. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374.

Du, X.; Liu, M.; Wang, K.; Wang, H.; Liu, J.; Chen, Y.; Feng, J.; Sha, C.; Peng, X.; and Lou, Y. 2023. ClassEval: A Manually-Crafted Benchmark for Evaluating LLMs on Class-level Code Generation. arXiv:2308.01861.

Farchi, E.; Froimovich, S.; Katan, R.; and Raz, O. 2024. Automatic Generation of Benchmarks and Reliable LLM Judgment for Code Tasks. arXiv:2410.21071.

Hendrycks, D.; Basart, S.; Kadavath, S.; Mazeika, M.; Arora, A.; Guo, E.; Burns, C.; Puranik, S.; He, H.; Song, D.; and Steinhardt, J. 2021. Measuring Coding Challenge Competence With APPS. arXiv:2105.09938.

Hui, B.; Yang, J.; Cui, Z.; Yang, J.; Liu, D.; Zhang, L.; Liu, T.; Zhang, J.; Yu, B.; Lu, K.; Dang, K.; Fan, Y.; Zhang, Y.; Yang, A.; Men, R.; Huang, F.; Zheng, B.; Miao, Y.; Quan, S.; Feng, Y.; Ren, X.; Ren, X.; Zhou, J.; and Lin, J. 2024. Qwen2.5-Coder Technical Report. arXiv:2409.12186.

Jiang, J.; Wang, F.; Shen, J.; Kim, S.; and Kim, S. 2024a. A Survey on Large Language Models for Code Generation. arXiv:2406.00515.

Jiang, X.; Dong, Y.; Wang, L.; Fang, Z.; Shang, Q.; Li, G.; Jin, Z.; and Jiao, W. 2024b. Self-planning code generation with large language models. *ACM Transactions on Software Engineering and Methodology*, 33(7): 1–30.

Jimenez, C. E.; Yang, J.; Wettig, A.; Yao, S.; Pei, K.; Press, O.; and Narasimhan, K. 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? arXiv:2310.06770.

Li, Y.; Choi, D.; Chung, J.; Kushman, N.; Schrittwieser, J.; Leblond, R.; Eccles, T.; Keeling, J.; Gimeno, F.; Dal Lago, A.; et al. 2022. Competition-level code generation with alphacode. *Science*, 378(6624): 1092–1097.

Mu, F.; Shi, L.; Wang, S.; Yu, Z.; Zhang, B.; Wang, C.; Liu, S.; and Wang, Q. 2024. Clarifygpt: A framework for enhancing llm-based code generation via requirements clarification. *Proceedings of the ACM on Software Engineering*, 1(FSE): 2332–2354.

Olausson, T. X.; Inala, J. P.; Wang, C.; Gao, J.; and Solar-Lezama, A. 2024. Is Self-Repair a Silver Bullet for Code Generation? arXiv:2306.09896.

OpenAI; Achiam, J.; Adler, S.; Agarwal, S.; Ahmad, L.; Akkaya, I.; Aleman, F. L.; Almeida, D.; Altenschmidt, J.; Altman, S.; Anadkat, S.; Avila, R.; Babuschkin, I.; Balaji, S.; Balcom, V.; Baltescu, P.; Bao, H.; Bavarian, M.; Belgum, J.; Bello, I.; Berdine, J.; Bernadett-Shapiro, G.; Berner, C.; Bogdonoff, L.; Boiko, O.; Boyd, M.; Brakman, A.-L.; Brockman, G.; Brooks, T.; Brundage, M.; Button, K.; Cai, T.; Campbell, R.; Cann, A.; Carey, B.; Carlson, C.; Carmichael, R.; Chan, B.; Chang, C.; Chantzis, F.; Chen, D.; Chen, S.; Chen, R.; Chen, J.; Chen, M.; Chess, B.; Cho, C.; Chu, C.; Chung, H. W.; Cummings, D.; Currier, J.; Dai, Y.; Decareaux, C.; Degry, T.; Deutsch, N.; Deville, D.; Dhar, A.; Dohan, D.; Dowling, S.; Dunning, S.; Ecoffet, A.; Eleti, A.; Eloundou, T.; Farhi, D.; Fedus, L.; Felix, N.; Fishman, S. P.; Forte, J.; Fulford, I.; Gao, L.; Georges, E.; Gibson, C.; Goel, V.; Gogineni, T.; Goh, G.; Gontijo-Lopes, R.; Gordon, J.; Grafstein, M.; Gray, S.; Greene, R.; Gross, J.; Gu, S. S.; Guo, Y.; Hallacy, C.; Han, J.; Harris, J.; He, Y.; Heaton, M.; Heidecke, J.; Hesse, C.; Hickey, A.; Hickey, W.; Hoeschele, P.; Houghton, B.; Hsu, K.; Hu, S.; Hu, X.; Huizinga, J.; Jain, S.; Jain, S.; Jang, J.; Jiang, A.; Jiang, R.; Jin, H.; Jin, D.; Jomoto, S.; Jonn, B.; Jun, H.; Kaftan, T.; Łukasz Kaiser; Kamali, A.; Kanitscheider, I.; Keskar, N. S.; Khan, T.; Kilpatrick, L.; Kim, J. W.; Kim, C.; Kim, Y.; Kirchner, J. H.; Kiros, J.; Knight, M.; Kokotajlo, D.; Łukasz Kondraciuk; Kondrich, A.; Konstantinidis, A.; Kosic, K.; Krueger, G.; Kuo, V.; Lampe, M.; Lan, I.; Lee, T.; Leike, J.; Leung, J.; Levy, D.; Li, C. M.; Lim, R.; Lin, M.; Lin, S.; Litwin, M.; Lopez, T.; Lowe, R.; Lue, P.; Makanju, A.; Malfacini, K.; Manning, S.; Markov, T.; Markovski, Y.; Martin, B.; Mayer, K.; Mayne, A.; McGrew, B.; McKinney, S. M.; McLeavey, C.; McMillan, P.; McNeil, J.; Medina, D.; Mehta, A.; Menick, J.; Metz, L.; Mishchenko, A.; Mishkin, P.; Monaco, V.; Morikawa, E.; Mossing, D.; Mu, T.; Murati, M.; Murk, O.; Mély, D.; Nair, A.; Nakano, R.; Nayak, R.; Neelakantan, A.; Ngo, R.; Noh, H.; Ouyang, L.; O'Keefe, C.; Pachocki, J.; Paino, A.; Palermo, J.; Pantuliano, A.; Parascandolo, G.; Parish, J.; Parparita, E.; Passos, A.; Pavlov, M.; Peng, A.; Perelman, A.; de Avila Belbute Peres, F.; Petrov, M.; de Oliveira Pinto, H. P.; Michael; Pokorny; Pokrass, M.; Pong, V. H.; Powell, T.; Power, A.; Power, B.; Proehl, E.; Puri, R.; Radford, A.; Rae, J.; Ramesh, A.; Raymond, C.; Real, F.; Rimbach, K.; Ross, C.; Rotsted, B.; Roussez, H.; Ryder, N.; Saltarelli, M.; Sanders, T.; Santurkar, S.; Sastry, G.; Schmidt, H.; Schnurr, D.; Schulman, J.; Selsam, D.; Sheppard, K.; Sherbakov, T.; Shieh, J.; Shoker, S.; Shyam, P.; Sidor, S.; Sigler, E.; Simens, M.; Sitkin, J.; Slama, K.; Sohl, I.; Sokolowsky, B.; Song, Y.; Staudacher, N.; Such, F. P.; Summers, N.; Sutskever, I.; Tang, J.; Tezak, N.; Thompson, M. B.; Tillet, P.; Tootoonchian, A.; Tseng, E.; Tuggle, P.; Turley, N.; Tworek, J.; Uribe, J. F. C.; Vallone, A.; Vijayvergiya, A.; Voss, C.; Wainwright, C.; Wang, J. J.; Wang, A.; Wang, B.; Ward, J.; Wei, J.; Weinmann, C.; Welihinda, A.; Welinder, P.; Weng, J.; Weng, L.; Wiethoff, M.; Willner, D.; Winter, C.; Wolrich, S.; Wong, H.; Workman, L.; Wu, S.; Wu, J.; Wu, M.; Xiao, K.; Xu, T.; Yoo, S.; Yu, K.; Yuan, Q.; Zaremba, W.; Zellers, R.; Zhang, C.; Zhang, M.; Zhao, S.; Zheng, T.; Zhuang, J.; Zhuk, W.; and Zoph, B. 2024. GPT-4 Technical Report. arXiv:2303.08774.

Rahmani, H. A.; Wang, X.; Feng, Y.; Zhang, Q.; Yilmaz, E.; and Lipani, A. 2023. A Survey on Asking Clarification Questions Datasets in Conversational Systems. In Rogers, A.; Boyd-Graber, J.; and Okazaki, N., eds., *Proceedings* *of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2698–2716. Toronto, Canada: Association for Computational Linguistics.

Shi, Z.; Feng, Y.; and Lipani, A. 2022. Learning to Execute Actions or Ask Clarification Questions. arXiv:2204.08373.

Team, G.; Georgiev, P.; Lei, V. I.; Burnell, R.; Bai, L.; Gulati, A.; Tanzer, G.; Vincent, D.; Pan, Z.; Wang, S.; et al. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530*.

Yang, A.; Yang, B.; Hui, B.; Zheng, B.; Yu, B.; Zhou, C.; Li, C.; Li, C.; Liu, D.; Huang, F.; Dong, G.; Wei, H.; Lin, H.; Tang, J.; Wang, J.; Yang, J.; Tu, J.; Zhang, J.; Ma, J.; Xu, J.; Zhou, J.; Bai, J.; He, J.; Lin, J.; Dang, K.; Lu, K.; Chen, K.; Yang, K.; Li, M.; Xue, M.; Ni, N.; Zhang, P.; Wang, P.; Peng, R.; Men, R.; Gao, R.; Lin, R.; Wang, S.; Bai, S.; Tan, S.; Zhu, T.; Li, T.; Liu, T.; Ge, W.; Deng, X.; Zhou, X.; Ren, X.; Zhang, X.; Wei, X.; Ren, X.; Fan, Y.; Yao, Y.; Zhang, Y.; Wan, Y.; Chu, Y.; Liu, Y.; Cui, Z.; Zhang, Z.; and Fan, Z. 2024. Qwen2 Technical Report. *arXiv preprint arXiv:2407.10671*.

Zamani, H.; Lueck, G.; Chen, E.; Quispe, R.; Luu, F.; and Craswell, N. 2020. MIMICS: A Large-Scale Data Collection for Search Clarification. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, CIKM '20, 3189–3196. New York, NY, USA: Association for Computing Machinery. ISBN 9781450368599.

Zhuo, T. Y.; Vu, M. C.; Chim, J.; Hu, H.; Yu, W.; Widyasari, R.; Yusuf, I. N. B.; Zhan, H.; He, J.; Paul, I.; Brunner, S.; Gong, C.; Hoang, T.; Zebaze, A. R.; Hong, X.; Li, W.-D.; Kaddour, J.; Xu, M.; Zhang, Z.; Yadav, P.; Jain, N.; Gu, A.; Cheng, Z.; Liu, J.; Liu, Q.; Wang, Z.; Lo, D.; Hui, B.; Muennighoff, N.; Fried, D.; Du, X.; de Vries, H.; and Werra, L. V. 2024. BigCodeBench: Benchmarking Code Generation with Diverse Function Calls and Complex Instructions. arXiv:2406.15877.

Zou, J.; Sun, A.; Long, C.; Aliannejadi, M.; and Kanoulas, E. 2023. Asking Clarifying Questions: To benefit or to disturb users in Web search? *Inf. Process. Manage.*, 60(2).

## A. LLM Prompts

### Question Generation

The prompt is presented below. For LLMs that do not support system prompts (e.g., Gemini), we incorporate the information typically included in the system prompt into the user prompt.

**Prompt for code generation task**

**System Prompt:**
You are an expert programmer. Your task is to complete the code given an incomplete code snippet. The code snippet contains the function's description. Before general the code, you can ask some questions about this functions.

**User Prompt:**
{**Coding task instruction**}
Before general the code, you can ask some questions about this functions. Olny generate the questions if you have questions about the task, do not make assumptions and generate code. Each question should be numbered with a heading, formatted as follows:
1. Question text
2. Question text
3. Question text

Generate code if you are confident that the information can guide you generate correct code.

### Answer Generation

The prompt is presented below. In the prompt, the LLM is instructed to avoid disclosing any implementation details in its responses.

**Prompt for answer generation task**

**System Prompt:**
You are an expert software architect. You are helping a programmer finalize a code snippet by answering questions from a software requirements perspective. For each provided question, you must produce your answer strictly in the following format:
Q: [The given question]
A: [Your answer]

Q: [The given question]
A: [Your answer]

... and so on.

Your answers should describe the expected behavior of the function in accordance with the canonical code without revealing any detailed information about the canonical code itself. Only answer the questions based on the given information. Do not mention the canonical code or provide any hints or suggestions regarding its implementation.

**User Prompt:**
# CODE SNIPPET:
{**Coding task instruction**}
# CANONICAL CODE:
{**Canonical code**}
# QUESTIONS:
{**Questions**}

### Code Generation without Q&A

The prompt is presented below. The LLM is instructed to generate code based on the coding instructions.

**Prompt for code generation without Q&A pairs task**

**System Prompt:**
You are an expert programmer. Your task is to complete a code snippet based on the provided context and requirements. You will be given a code snippet that needs to be completed.

# Your completed code must:
1. Strictly adhere to the described behavior and requirements.
2. Be clear, concise, and maintainable, following programming best practices.
3. Avoid adding unnecessary features or deviating from the described requirements.
4. Handle any specified edge cases or constraints appropriately.
Ensure your implementation is robust and aligns with the functional requirements derived from the provided information.

**User Prompt:**
# CODE SNIPPET:
{**Coding task instruction**}
Based on the information, please generate the code.

## Code Generation with Q&A

The prompt is presented below. The LLM is instructed to generate code based on the coding instructions and additional Q&A pairs.

**Prompt for code generation with Q&A pairs task**

**System Prompt:**
You are an expert programmer. Your task is to complete a code snippet based on the provided context and requirements. You will be given:
1. A code snippet that needs to be completed.
2. A set of question-answer pairs describing the expected behavior of the function or program.

# Your completed code must:
1. Strictly adhere to the described behavior and requirements provided in the question-answer pairs.
2. Be clear, concise, and maintainable, following programming best practices.
3. Avoid adding unnecessary features or deviating from the described requirements.
4. Handle any specified edge cases or constraints appropriately.
Ensure your implementation is robust and aligns with the functional requirements derived from the provided information.

**User Prompt:**
# CODE SNIPPET:
{**Coding task instruction**}
# QUESTION-ANSWER PAIRS:
{**Q&A pairs**}
Based on the information, please generate the code.