ClarifyGPT: A Framework for Enhancing LLM-Based Code Generation via Requirements Clarification

FANGWEN MU[*†], State Key Laboratory of Intelligent Game, Institute of Software, CAS, China
LIN SHI[*], Beihang University, China
SONG WANG, York University, Canada
ZHUOHAO YU[†], State Key Laboratory of Intelligent Game, Institute of Software, CAS, China
BINQUAN ZHANG, Beihang University, China
CHENXUE WANG[‡], State Key Laboratory of Intelligent Game, Institute of Software, CAS, China
SHICHAO LIU, Software IDE innovation Lab, Huawei Central Software Institute, China
QING WANG[†§], State Key Laboratory of Intelligent Game, Institute of Software, CAS, China

Large Language Models (LLMs), such as ChatGPT, have demonstrated impressive capabilities in automatically generating code from provided natural language requirements. However, in real-world practice, it is inevitable that the requirements written by users might be ambiguous or insufficient. Current LLMs will directly generate programs according to those unclear requirements, regardless of interactive clarification, which will likely deviate from the original user intents. To bridge that gap, we introduce a novel framework named ClarifyGPT, which aims to enhance code generation by empowering LLMs with the ability to identify ambiguous requirements and ask targeted clarifying questions. Specifically, ClarifyGPT first detects whether a given requirement is ambiguous by performing a code consistency check. If it is ambiguous, ClarifyGPT prompts an LLM to generate targeted clarifying questions. After receiving question responses, ClarifyGPT refines the ambiguous requirement and inputs it into the same LLM to generate a final code solution. To evaluate our ClarifyGPT, we invite ten participants to use ClarifyGPT for code generation on two benchmarks: MBPP-sanitized and MBPP-ET. The results show that ClarifyGPT elevates the performance (Pass@1) of GPT-4 from 70.96% to 80.80% on MBPP-sanitized. Furthermore, to conduct large-scale automated evaluations of ClarifyGPT across different LLMs and benchmarks without requiring user participation, we introduce a high-fidelity simulation method to simulate user responses. The results demonstrate that ClarifyGPT can significantly enhance code generation performance compared to the baselines. In particular, ClarifyGPT improves the average performance of GPT-4 and ChatGPT across five benchmarks from 62.43% to 69.60% and

[*]Both authors contributed equally to this research
[†]Also With University of Chinese Academy of Sciences
[‡]Also With Harbin Institute of Technology
[§]Corresponding author

Authors' addresses: Fangwen Mu, fangwen2020@iscas.ac.cn, State Key Laboratory of Intelligent Game, Institute of Software, CAS, Beijing, China; Lin Shi, Beihang University, Beijing, China, shilin@buaa.edu.cn; Song Wang, York University, Toronto, Canada, wangsong@yorku.ca; Zhuohao Yu, State Key Laboratory of Intelligent Game, Institute of Software, CAS, Beijing, China, yuzhuohao23@mails.ucas.edu.cn; Binquan Zhang, Beihang University, Beijing, China, binquan@buaa.edu.cn; ChenXue Wang, State Key Laboratory of Intelligent Game, Institute of Software, CAS, Beijing, China, chenxuew02@gmail.com; Shichao Liu, Software IDE innovation Lab, Huawei Central Software Institute, Beijing, China, liushichao2@huawei.com; Qing Wang, State Key Laboratory of Intelligent Game, Institute of Software, CAS, Beijing, China, wq@iscas.ac.cn.

ACM 2994-970X/2024/7-ART103
https://doi.org/10.1145/3660810

from 54.32% to 62.37%, respectively. A human evaluation also confirms the effectiveness of ClarifyGPT in detecting ambiguous requirements and generating high-quality clarifying questions. We believe that ClarifyGPT can effectively facilitate the practical application of LLMs in real-world development environments.

CCS Concepts: • **Software and its engineering → Automatic programming**; • **Computing methodologies → Machine learning**.

Additional Key Words and Phrases: Code Generation, Large Language Model, Prompt Engineering

ACM Reference Format:
Fangwen Mu, Lin Shi, Song Wang, Zhuohao Yu, Binquan Zhang, ChenXue Wang, Shichao Liu, and Qing Wang. 2024. ClarifyGPT: A Framework for Enhancing LLM-Based Code Generation via Requirements Clarification. *Proc. ACM Softw. Eng.* 1, FSE, Article 103 (July 2024), 23 pages. https://doi.org/10.1145/3660810

1 INTRODUCTION

Code generation aims to produce a code snippet that satisfies the user's intent expressed in a natural language requirement. This task, offering potential cost savings, acceleration of programming activities, and facilitation of software development, has consequently garnered attention across various domains, e.g., natural language processing, artificial intelligence, and software engineering. Recent efforts tackle this task by leveraging Large Language Models (LLMs) with billions of parameters, such as ChatGPT [34] and CodeGen [33]. The LLMs take the natural language requirements (i.e., prompts) as inputs and output the corresponding code snippets, achieving remarkable progress in code generation.

However, in real-world practice, due to the diversity of user experience and perspective, it is inevitable that the requirements written by users might be ambiguous or insufficient. For example, the requirement *"Write a function to sort a list of elements"* does not specify whether the user intends for the list to be sorted in ascending or descending order. Current LLMs do not handle such ambiguous requirements: they rarely ask users to clarify these requirements and instead directly generate programs that may deviate from the users' needs [21]. Current LLMs-based code generation approaches lack the mechanism of clarifying unclear requirements [20, 21], i.e., they directly generate programs according to those unclear requirements regardless of interactive clarification. In contrast, when human developers encounter ambiguous requirements, they seek additional information by interactively asking clarifying questions to the users. For the above example, a simple clarifying question such as *"Should the sorting be in ascending or descending order?"* could help disambiguate the requirement.

In light of this observation, we argue that empowering LLMs with the ability to automatically ask clarifying questions for ambiguous requirements is necessary for improving the quality and efficiency of code generation. However, it is quite challenging to empower LLMs with this ability due to the following barriers. **(1) When to Ask Clarifying Questions?** In practical development environments, numerous requirements exist, including both ambiguous and unambiguous ones. Failure to concentrate on questioning only the ambiguous requirements can lead to unnecessary interactions between LLMs and users regarding well-defined requirements. These unnecessary interactions, in turn, can diminish efficiency and compromise the user experience. **(2) What Clarifying Questions Should be Asked?** The quality of clarifying questions also influences the efficiency and performance of code generation. Precise and targeted questions aid users in expressing their intents clearly, ensuring that the obtained responses are directly relevant to the ambiguities present in the requirements. Vague or broad questions increase the risk of obtaining off-topic or irrelevant responses, potentially hindering LLMs from comprehending user intents.

In this paper, we propose a novel framework called ClarifyGPT that enhances LLM-based code generation via requirements clarification. **First**, we employ a two-step code consistency check

to decide when to ask clarifying questions. We are motivated by the observation that feeding a clear requirement to LLMs usually results in generating diverse code snippets that behave consistently, i.e., given the same test inputs, those different code snippets will likely return the same outputs. While feeding an unclear requirement, LLMs are likely to generate diverse code snippets that behave differently. Specifically, in the first step, ClarifyGPT aims to generate numerous high-quality test inputs for a given requirement via type-aware mutation. In the second step, ClarifyGPT inputs the given requirement into an LLM to sample n code solutions and checks whether they produce identical outputs when tested with the generated input. If the outputs are not identical, ClarifyGPT determines that the requirement requires further clarification; and vice versa. **Second**, we employ the reasoning-based prompting for generating clarifying questions. Initially, ClarifyGPT directs LLMs to analyze the factors contributing to the ambiguity of the given requirement by comparing code solutions with different functionalities. Subsequently, it formulates targeted clarifying questions based on the results of this analysis. By comparing these different code implementations, potential points of ambiguity in the requirements can be readily identified. After detecting the points of ambiguity in the requirements, the LLMs can generate targeted clarifying questions for them. **Finally**, ClarifyGPT refines the original requirement based on the generated questions and their responses and generates the final code solution.

To assess the effectiveness of ClarifyGPT, we first integrate GPT-4 [35] into ClarifyGPT and recruit ten participants to evaluate its performance on two public benchmarks (MBPP-sanitized [5], and MBPP-ET [11]). The results show that ClarifyGPT elevates the performance (Pass@1) of GPT-4 on MBPP-sanitized from 70.96% to 80.80%, improves the performance (Pass@1) of GPT-4 on MBPP-ET from 51.52% to 60.19%. Besides, due to requiring the involvement of human participants, evaluating ClarifyGPT could be very expensive and hard to reproduce. To perform automated evaluations of ClarifyGPT across different LLMs and benchmarks without requiring user participation, we introduce a high-fidelity simulation method to simulate user feedback. We then conduct comprehensive experiments on five benchmarks (HumanEval [9], HumanEval-ET [11], MBPP-sanitized, MBPP-ET, and CoderEval [49]) using two state-of-the-art LLMs (i.e., GPT-4 and ChatGPT). The results demonstrate that, in comparison with the default GPT-4, ClarifyGPT achieves an average improvement of 11.66% across five benchmarks; in comparison with the default ChatGPT, ClarifyGPT achieves an average improvement of 15.00% on five benchmarks. We also conduct a human evaluation to assess the effectiveness of ClarifyGPT in detecting ambiguous requirements and generating high-quality clarifying questions, showcasing ClarifyGPT's commendable effectiveness in both tasks. Our main contributions are outlined as follows:

- **Framework**: We propose a novel framework, named ClarifyGPT, which enables LLMs to detect ambiguous requirements and formulate targeted clarifying questions. ClarifyGPT refines the ambiguous requirements based on the answers to clarifying questions and further generates code solutions.
- **User Simulation**: We introduce a user simulation method for producing high-fidelity simulated answers to the clarifying questions, which facilitates automated evaluations of ClarifyGPT across different LLMs and benchmarks, eliminating the necessity for direct user participation.
- **Evaluation**: We conduct extensive experiments on five widely-used benchmarks to show that, ClarifyGPT achieves substantial improvements across different models and benchmarks. A human evaluation further confirms the significant potential of applying ClarifyGPT in real-world practice.
- **Data**: publicly accessible dataset and source code [2] to facilitate the replication of our study and its application in extensive contexts.

2 BACKGROUND AND RELATED WORK

2.1 LLM-Based Code Generation

Code generation is a hot research topic for software engineering and artificial intelligence communities. Recently, many LLMs have been proposed for code generation. One class of models is the encoder-decoder models, e.g., PLBART [3], CodeT5 [45], and AlphaCode [27], which generally encode an input text into a context embedding and decode the embedding to a code solution. Another class of models is the decoder-only models that are trained with the next token prediction objective and generate code from left to right. GPT series models [6, 9], PolyCoder [48], and InCoder [14] are examples of such models. Among them, ChatGPT [34] and GPT-4 [35] are the state-of-the-art LLMs developed by OpenAI. They have demonstrated improved understanding and reasoning abilities, proficiency in comprehending the provided context, and the capacity to generate high-quality texts.

Since training or fine-tuning these LLMs is highly expensive, there has also been a lot of research focused on enhancing the performance of LLMs in code generation with minimal or no fine-tuning. Prompt Learning is one of the most important techniques for achieving this goal [12, 28, 31, 40, 47]. The Chain-of-Thought (CoT) [47] is a novel prompting engineering technique, which can elicit LLMs to produce intermediate reasoning steps that lead to the final answer. It has shown impressive performance in complex reasoning tasks (e.g., arithmetic and symbolic reasoning) [19, 47], and has therefore been applied to code generation [17, 26]. Inspired by CoT, Li et al. [26] propose a new prompting method, named Structured CoT (SCoT). Different from CoT, SCoT explicitly introduces code structures and teaches LLMs to generate intermediate reasoning steps with program structures. Jiang et al. [17] propose a self-planning approach that can guide LLMs to understand code planning with few-shot demonstrations and write corresponding code planning for the given requirement. The aforementioned studies focus on leveraging and augmenting the reasoning capabilities of LLMs, that is, prompting LLMs to generate intermediate reasoning steps to enhance code generation performance. Nevertheless, they remain insufficient in addressing the ambiguous requirements provided by humans, as unclear user intent may mislead LLMs into producing incorrect reasoning steps, thereby yielding inaccurate results. Our ClarifyGPT recognizes the importance of clarifying ambiguous requirements and proposes a novel framework that enables LLMs to automatically detect ambiguous requirements and ask targeted clarifying questions. By clarifying user requirements, ClarifyGPT can generate code solutions that fulfill the user's intentions. GPT-Engineer [36] is a recent open-source GitHub repository. It utilizes manual-designed instructions to prompt LLMs to ask clarifying questions for the input user requirements, and then generates code snippets based on user feedback. However, GPT-Engineer asks clarifying questions for both ambiguous and unambiguous requirements, which is detrimental to the user experience and may result in incorrect code solutions[1]. In contrast, ClarifyGPT can detect ambiguous requirements by checking whether the test outputs of sampled code solutions are identical. Furthermore, ClarifyGPT employs prompting techniques to direct LLMs to first analyze the factors contributing to requirement ambiguity and then formulate targeted questions.

2.2 Clarifying Question Generation

The task of generating clarifying questions for ambiguous queries or dialogues has received much attention in information retrieval and dialogue system fields [10, 18, 20, 30, 37, 41]. In terms of information retrieval, many studies have pointed out that clarifying questions can help resolve ambiguous queries and improve user experience. For example, Wang and Li [43] find that search queries are often short and the underlying user intents are often ambiguous. They propose an effective template-guided clarifying question generation model, which employs Transformer to

[1] https://github.com/AntonOsika/gpt-engineer/issues/708

select a question template from a list of template candidates and fill in the question slot from a slot vocabulary. Eberhart and McMillan [13] propose a novel method to ask clarifying questions for query refinement, which utilizes a task extraction algorithm to identify query aspects and follows a rule-based procedure to generate questions. In terms of the dialogue system domain, both rule-based and learning-based approaches have been proposed. Dhole [10] proposes a novel method of generating discriminative questions by leveraging a simple rule-based system, which aims at seeking clarification from the user, thereby reducing the roboticity of the conversation and making the interaction considerably natural. Rao et al. [37] describe a method for generating clarifying questions, which uses a seq2seq model to generate a question given a context and utilizes another seq2seq model to generate an answer given the context and the question.

In code generation, dealing with ambiguous user requirements has received little attention so far. To the best of our knowledge, Li et al. [25] is the only research paper that addresses ambiguous requirements resolution for code generation. This work aims to clarify the ambiguous requirements missing key operations, e.g., API calls. It first collects a dataset named Code ClarQA containing natural language requirements, code, clarifying questions, and answers. Then, it proposes a code generation pipeline that can select relevant clarifying questions and their answers from the dataset for a given requirement for generating a code solution. However, the scope of applicability for this work is limited. Firstly, it primarily focuses on clarifying operational-level ambiguities, leaving other forms of ambiguity, such as semantic ambiguities in natural language requirements, less effectively addressed. Furthermore, it heavily relies on the constructed dataset, retrieving relevant questions for ambiguous requirements. If the dataset lacks similar requirements, the method's performance may suffer. Differing from this work, ClarifyGPT is not limited to a specific type of ambiguous requirements clarification. ClarifyGPT can generate precise and targeted questions for various requirements by leveraging the powerful understanding ability of LLMs.

3 APPROACH

In this section, we introduce ClarifyGPT, a code generation framework for LLMs. Figure 1 illustrates the overview of ClarifyGPT, which consists of four main stages: (1) **Test Input Generation** (Section 3.1), aiming at generating high-quality test inputs for a given requirement by using prompting techniques and heuristic mutations; (2) **Code Consistency Check** (Section 3.2), for leveraging the generated test inputs to conduct a consistency evaluation, and then identifying the ambiguous requirements; (3) **Reasoning based question generation** (Section 3.3), focused on generating targeted clarifying questions for the identified ambiguous requirements by prompting LLMs to engage in intermediate reasoning; (4) **Enhanced Code Generation** (Section 3.4), which incorporates the clarifying questions and their feedback to refine the original requirement and generate the final code solution based on the refined prompt. Below, we provide details for each stage in ClarifyGPT.

3.1 Test Input Generation

In this step, ClarifyGPT aims to produce high-quality test inputs to effectively distinguish between code solutions with different functionalities. There are many studies have attempted to employ LLMs for unit test case generation [24, 38, 42] and have demonstrated impressive performance. Following prior work [29], ClarifyGPT leverages LLMs as the test input generator and generates test inputs by adopting a two-step approach (i.e., seed input initialization and type-aware mutation). Specifically, ClarifyGPT begins by designing a prompt to instruct an LLM in creating a set of seed inputs. It then performs type-aware mutations to generate a large number of new inputs. Our insights are twofold: (1) on one hand, since LLMs possess powerful understanding and reasoning abilities, using them as test input generators can produce high-quality inputs that remain valid

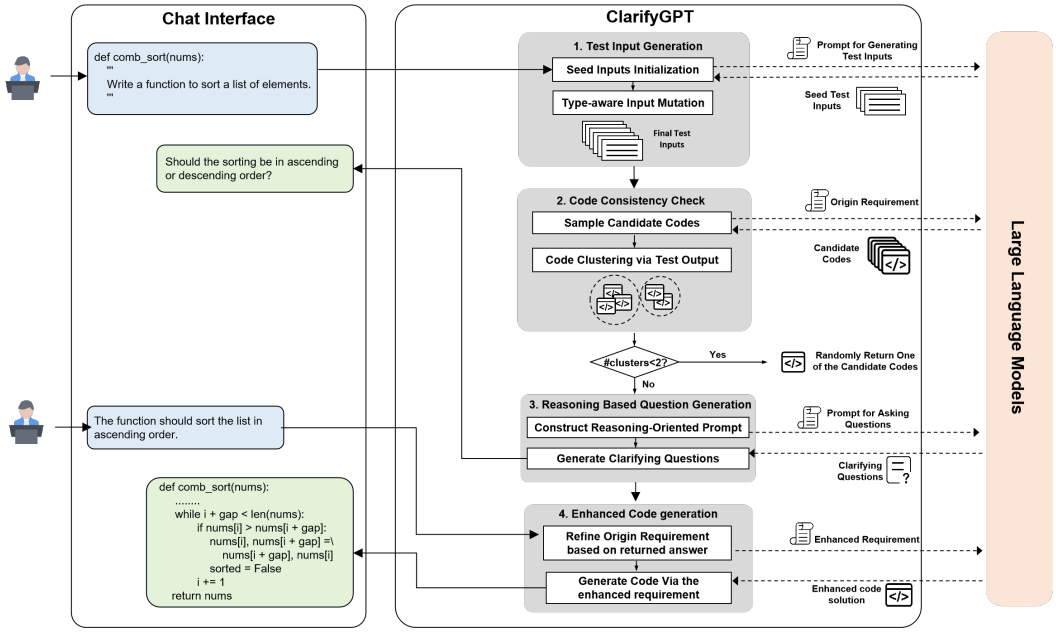

Fig. 1. The Overview of ClarifyGPT

even under semantic constraints. Traditional input generators often face challenges in ensuring compliance with such semantic constraints. (2) on the other hand, LLMs are unsuitable for large amounts of automated test generation due to undesired speed and cost of querying such large models [29]. Thus, we utilize a heuristic mutation-based method to accelerate the generation of numerous test cases, ensuring both stability and reliability.

3.1.1 Seed Input Initialization. ClarifyGPT starts with designing a prompt for seed input initialization. As shown in Figure 2 (a), the prompt consists of three parts: (1) an instruction, designed to elicit LLMs to generate complex, difficult, and corner-case test inputs; (2) few-shot manually-crafted demonstrations, including a user requirement and ground-truth test inputs, which can assist LLMs in better understanding the task described in the instruction; (3) a query, for which LLMs generate input tests based on it. Specifically, we first finalize the prompt with the instruction, demonstrations, and the given requirement. Then, ClarifyGPT utilizes the prompt to query LLMs to generate seed inputs. Finally, we collect these generated seed inputs to initialize a seed pool that will be used for mutation.

3.1.2 Type-Aware Input Mutation. After initializing a seed pool, ClarifyGPT employs a type-aware input mutation strategy [29] to generate higher-quality test inputs. Specifically, our approach follows the standard mutation-based fuzzing workflow [50, 51]: (1) At each iteration, an input is randomly selected from the seed pool. (2) For the selected input, we inspect its data types and perform a single mutation operation consistent with its type to create a new test case. The basic mutations used for different types of inputs are illustrated in Table 1. For simple data types, such as *int* and *float*, one mutation operation simply increases or decreases its value by 1. For compound types, we mutate the elements based on their internal types. (3) After completing a round of mutations, we add the newly generated inputs to the seed pool and repeat the aforementioned process until we attain the desired number of generated inputs.

Table 1. List of basic type-aware mutations over input x [29]

Type	Mutation	Type	Mutation
$int\|float$	Returns $x \pm 1$	$List$	Remove/repeat a random item $x[i]$ Insert/replace $x[i]$ with Mutate($x[i]$)
$bool$	Returns a random boolean	$Tuple$	Returns $Tuple$(Mutate($List(x)$))
$NoneType$	Returns $None$	Set	Returns Set(Mutate($List(x)$))
str	Remove a substring s Repeat a substring s Replace s with Mutate(s)	$Dict$	Remove a key/value pair $k \rightarrow v$ Update $k \rightarrow v$ to $k \rightarrow$ Mutate(v) Insert Mutate(k) $\rightarrow$ Mutate(v)

3.2 Code Consistency Check

A clear user requirement should be easy to understand and leave no room for interpretation, while an ambiguous user requirement can lead to stakeholders interpreting it in different ways. Inspired by this, we make an assumption that, for a given requirement, if an LLM generates numerous code solutions with different functionalities, it signifies that the requirement can lead to the LLM interpreting it in different ways. Consequently, such a requirement necessitates further clarification and refinement. In light of this assumption, we propose a simple yet efficient method to determine ambiguous requirements. First, we feed a given requirement into an LLM to sample n code solutions. Then, these code solutions are executed with test inputs generated in the previous step. We obtain the test outputs of these programs and compare the test outputs to inspect whether they are identical. If the outputs are identical, ClarifyGPT considers these code solutions as interpreting the requirement in the same way, thus identifying the requirement as unambiguous. In this case, one of the sampled codes would be output as the final code solution. However, if the outputs are not identical, ClarifyGPT believes the LLM has different understandings of this requirement when it produces code solutions and identifies the requirement as ambiguous. For these ambiguous requirements, as shown in Figure 1, we perform code clustering, which involves dividing these code solutions into several groups based on their test outputs. Subsequently, ClarifyGPT randomly chooses one code solution from each group and feeds these inconsistent code solutions into the next component to synthesize the prompt used for asking questions.

3.3 Reasoning Based Question Generation

Targeted clarifying questions facilitate users in articulating their intentions with clarity, ensuring that the responses obtained are directly pertinent to the unclear parts within the requirements. Vague or broad questions increase the risk of getting off-topic or irrelevant responses, which may hurt the performance of code generation. Therefore, upon identifying ambiguous requirements, it becomes essential to empower LLMs with the capability to craft precise and targeted questions. To achieve this objective, we devise a reasoning-based prompt aimed at directing LLMs to initially scrutinize the factors contributing to the ambiguity of the requirement and subsequently formulate targeted questions grounded in the analysis. The designed prompt is depicted in Figure 2 (b). It includes three parts: (1) an instruction, which describes the task (i.e., clarifying question generation) we want the LLMs to solve; (2) few-shot <requirement, inconsistent solutions, clarifying questions> triples as demonstrations, which help LLMs in understanding and solving the task; (3) a query, containing a user requirement and its code solutions, which is fed to LLMs for generating questions.

Specifically, ClarifyGPT constructs the prompt to direct LLMs to analyze the factors contributing to the unclear requirement by understanding the functionalities of these inconsistent code solutions and comparing their disparities. The motivation is that, in software development, code solutions

Fig. 2. The details of the prompts used in ClarifyGPT

represent the specific implementation of requirements. If a requirement is ambiguous, different developers may have different interpretations and consequently write different code. Some of these inconsistent code solutions are incorrect or not in line with the original intent. By comparing these different code implementations, potential points of ambiguity in the requirements can be easily identified. After detecting the points of ambiguity in the requirements, the LLMs continue to generate targeted clarifying questions based on the detection results.

Our proposed prompting shares a similar idea with the Chain of Thought (CoT) [47] prompting, which elicits LLMs to generate intermediate reasoning steps (analysis of the factors contributing to ambiguity) first, and then produce final results (targeted clarifying questions) based on these intermediate reasoning steps. In this way, ClarifyGPT encourages LLMs to perform "far-ahead

planning" [7], enabling them to better leverage their reasoning and comprehension abilities to enhance the quality of the generated questions.

3.4 Enhanced Code Generation

Once user responses are captured, ClarifyGPT combines them with the generated questions to refine the original requirement into a clear one. In particular, as shown in the query in Figure 2 (d), we pair each question and its corresponding answer to create a clarification, which are then appended to the end of the docstring to form the refined requirement. By refining an ambiguous requirement in this way, we can preserve the structural integrity of the docstring in the original requirement while enhancing it with additional clarifying information. Subsequently, we use the refined requirement to construct a prompt to instruct LLMs in generating the final code solution. The constructed prompt also consists of three parts, i.e., an instruction, some demonstrations, and a query, as depicted in Figure 2 (d).

4 EXPERIMENTAL DESIGN

To evaluate the effectiveness of ClarifyGPT, we conduct comprehensive experiments. In this section, we illustrate our experimental design, including research questions, models, benchmarks, metrics, baselines, and implementation details.

4.1 Research Questions

We address the following three research questions to assess the performance of ClarifyGPT.

RQ1: How does the ClarifyGPT perform when receiving real user feedback in comparison to baseline approaches? In real-world scenarios, ClarifyGPT assists users in writing code by interacting with them, i.e., asking for clarification and receiving user feedback. Thus, in this RQ, we explore whether ClarifyGPT with humans in the loop can achieve higher performance than existing code generation baselines. Since evaluating interactive code generation with human participants is costly, we only select GPT-4 as the base model, and hire ten participants (including academic researchers and industry developers) to manually answer the clarifying questions generated by ClarifyGPT. We compare ClarifyGPT to three baselines on two benchmarks (i.e., MBPP-sanitized and MBPP-ET).

RQ2: How does the ClarifyGPT perform when receiving simulated user feedback compared to the state-of-the-art baseline approaches? This RQ performs large-scale automated evaluations of ClarifyGPT across different LLMs and benchmarks without requiring user participation, which aims to further verify whether ClarifyGPT can achieve higher performance than existing code generation baselines. We first propose a user simulation method that leverages LLMs to simulate user feedback. Then, we apply three baselines and ClarifyGPT to two representative LLMs (i.e., GPT-4 and ChatGPT), and evaluate their performance on five widely-used benchmarks (i.e., HumanEval, MBPP-sanitized, HumanEval-ET, MBPP-ET, and CoderEval).

RQ3: How does the number of demonstrations in a prompt impact the performance of ClarifyGPT? Prompting techniques could be sensitive to the number of demonstrations [15, 32]. In this research question, we measure the performance of ClarifyGPT with varying numbers of demonstrations to investigate the prompt robustness of ClarifyGPT.

4.2 Studied LLMs

There are many LLMs available for code generation. However, the specific context of this work necessitates that the LLMs possess a certain level of communicative competence, that is, the ability to comprehend human instructions and formulate clarifying questions. Thus, the LLMs without instruction tuning (e.g., InCoder [14] and CodeGen [33]) are not suitable as the base models applied

to ClarifyGPT framework. In this work, we select two representative chat-LLMs (i.e., ChatGPT and GPT4) as base models to evaluate ClarifyGPT framework.

- **ChatGPT** [34] is one of the most powerful chat models empowered by OpenAI. It is trained using a novel approach called Reinforcement Learning from Human Feedback (RLHF), which seamlessly integrates reinforcement learning and human feedback. Specifically, ChatGPT is first trained with vast amounts of natural language text and code files. Then, it is fine-tuned through reinforcement learning, enabling it to adeptly comprehend and execute human instructions. In our experiments, We use OpenAI's API to access the ChatGPT model, i.e., gpt-3.5-turbo.
- **GPT-4** [35] is OpenAI's most advanced LLM, which can accept image and text inputs, emit text outputs. It is also trained with reinforcement learning and learns to follow human instructions. GPT-4 has demonstrated improved language understanding, allowing it to comprehend complex and nuanced contexts, making it highly effective on many downstream tasks, including text summarization, translation, and code generation [7]. In our experiments, we use OpenAI's API to access the GPT-4 model, i.e., gpt-4-turbo.

4.3 Benchmarks

Following the previous work [8, 12, 17, 26], we select public code generation benchmarks, namely HumanEval [9], MBPP-sanitized [5], along with their extended test case versions (i.e., HumanEval-ET and MBPP-ET [11]) for our experimental evaluation. Additionally, to assess ClarifyGPT's efficacy in practical development contexts, we include experiments on a pragmatic code generation benchmark, CoderEval [49]. The statistics of these benchmarks are shown in Table 2.

- **HumanEval** [9] is a hand-written problem-solving dataset crafted subsequent to the cut-off date of Codex's training dataset, consisting of 164 Python programming problems. Programming problems in the HumanEval concern language comprehension, algorithms, and mathematics. Each problem includes a function signature, a natural language requirement, and several unit tests. A problem is considered solved by code-LLMs when all unit tests are passed.
- **MBPP-sanitized** [5] is a hand-verified subset of MBPP (Mostly Basic Programming Problems) dataset, which contains 427 crowd-sourced Python programming problems, involving numeric manipulations, standard libraries functionality, and more. Each problem contains a function signature, a user requirement, and three test cases.
- **HumanEval-ET** and **MBPP-ET** [11] are two extended versions of HumanEval and MBPP benchmarks with an average of 100+ additional test cases per problem. To improve the reliability of generated code evaluation, they collect many edge test cases that are not included in original benchmarks.
- **CoderEval** [49] is a benchmark of pragmatic code generation. Compared with the widely used HumanEval and MBPP benchmarks, CoderEval includes non-standalone functions that are collected from various open-source projects. We evaluate ClarifyGPT on the Java version of CoderEval, which contains 230 Java programming tasks. Specifically, given that ClarifyGPT requires generating test inputs for code consistency check, while accurately generating non-built-in types of inputs (such as user-defined variables or objects) poses a challenge for current LLMs and is not the scenario that this paper focuses on. Therefore, we exclude the programming tasks with non-built-in types of input arguments in CoderEval, leaving 163 programming tasks for evaluation. To assess the performance of different methods on CoderEval, we use an evaluation platform [1], which provides a ready runtime environment with automatic programs to verify the code generated by code models.

Table 2. Statistics of benchmarks: the total number of problems in the benchmark (Problem Nums), the average number of test cases per problem (AVG.Tests per Problem), and the average/maximum/minimum number of prompt words in the benchmark (AVG/MAX/MIN.Words in Prompt).

Benchmark	HumanEval	HumanEval-ET	MBPP-sanitized	MBPP-ET	CoderEval
Problem Nums	164	164	427	427	163
AVG.Tests per Problem	7.8	107.5	3.1	101.7	-
AVG.Words in Prompt	67.7	67.7	14.5	14.5	42.84
MAX.Words in Prompt	249	249	47	47	103
MIN.Words in Prompt	17	17	7	7	8

4.4 Evaluation Metrics

We evaluate the accuracy of the generated code using the metric Pass@k [22]. This metric serves as an estimator of the generational capabilities under a specific budget, which is widely used in previous LLM-related studies [8, 23, 52]. For each problem in the benchmarks, we generate k code solutions, and if any of the k code solutions passes all tests, this problem is considered solved. In real-world development scenarios, generating k code will impose a burden on developers, that is, they need to read and understand k different code and select one as the target code. Thus, in this paper, the k is set to 1, which satisfies most scenarios where developers consider only single-generated code [12, 17]. To avoid high variance and randomness, we run each approach three times and report the average results as the final results.

4.5 Comparison Baselines

- **Default LLM:** takes the original requirements directly from benchmarks as inputs to prompt LLMs for code generation.
- **CoT (Chain-of-Thought)** [47]: generates a series of reasoning steps for each requirement by using the CoT prompt and then generates the corresponding code. To ensure the fairness of comparison, the CoT baseline has the same number of demonstrations (i.e., three demonstrations) and demonstration seeds.
- **GPT-Engineer**[2]: is a recent open-source GitHub repository. It utilizes manual-designed instructions to elicit LLMs to ask clarifying questions for the input user requirements and then generates code snippets based on user feedback.

4.6 Implementation Details

The implementation details of constructing prompts and configuring models in ClarifyGPT are as follows.

Prompt Construction. Following previous work [44, 47], we select the first three problems from each benchmark and extract the user requirements from these problems as demonstration seeds. We use the remaining problems (except for the first three problems) from each benchmark as the test set. Subsequently, we manually create distinct demonstrations for various prompts, as illustrated in Figure 2. It should be noted that the reason we only create three demonstrations for each prompt is due to the input length limit of LLMs.

Model Configuration. We treat the two LLMs used in the experiments as black box generators and only set a few interface parameters they provide without accessing internal parameters. For all LLMs, we set the *top p* to 0.95, the *frequency_penalty* to 0. The *max_tokens* represents the maximum number of tokens to be generated, which is set to 800 for the prompt of asking clarifying questions

[2]https://github.com/AntonOsika/gpt-engineer

and 300 for other prompts. In particular, we set the *temperature* to 0, except when sampling code solutions, for which the *temperature* is set to 0.8. We follow Chen et al. [9] to truncate the generated content generated by five stop sequences: "\nclass", "\ndef", "\n#", "\nif", and "\nprint".

5 RESULTS AND ANALYSIS

5.1 RQ1: Performance with Real User Feedback

Setup. In this RQ, we explore how ClarifyGPT performs in real-world scenarios, that is, whether ClarifyGPT can achieve higher performance than existing code generation baselines when receiving real user feedback. Specifically, we apply ClarifyGPT to the GPT-4 model. Since MBPP-ET benchmark shares the same user requirements as MBPP-sanitized, we only apply ClarifyGPT to the original versions of the benchmark (i.e., MBPP-sanitized) and report ClarifyGPT's performance on these two benchmarks using their respective unit tests. ClarifyGPT first takes the user requirement of each problem in the benchmarks as input and determines them as ambiguous or unambiguous. Then, it generates clarifying questions for the ambiguous requirements. In total, ClarifyGPT identified 287 problems with unambiguous requirements and 140 problems with ambiguous requirements from MBPP-sanitized benchmark. We collected these 140 ambiguous problems along with their clarifying questions generated by ClarifyGPT. The average number of clarifying questions per problem is 2.85. We crafted three identical questionnaires for each problem, ensuring that each problem would be assessed by three different participants. Each questionnaire consists of three elements: (1) the (ambiguous) requirement of the problem, which describes the problem's intent; (2) the unit test cases containing expected input-output examples, which assist participants in understanding the problem's intent; (3) the generated clarifying questions, which participants are required to answer.

We recruited ten participants, including three Ph.D. students, two Master's students, two senior researchers, and three industry developers. None of them are co-authors of this paper. All participants have at least three years of experience in Python development, with six of them having more than five years of experience. Participants were initially provided with task descriptions and example questionnaires that contained appropriate question answers. After completing a training exercise, we assigned 42 problems to each participant and asked them to respond to the clarifying questions based on the information provided in the questionnaires. Each problem will be solved by three participants.

We collected the answers provided by the participants and input them into ClarifyGPT to generate final code solutions. As mentioned earlier, we evaluated the correctness of the generated code on the two benchmarks using the unit test cases. Since each problem's clarifying questions were answered by three participants, we report the average Pass@1 results.

Results. The comparison results between the performance of ClarifyGPT receiving human feedback and other baselines are depicted in Table 3. The values in red are ClarifyGPT (Human Feedback)'s relative improvements compared to the *Default* baseline.

We can see that ClarifyGPT (Human Feedback) achieves the best performance on the two benchmarks. Compared with the *Default*, ClarifyGPT (Human Feedback) exhibits significant improvements in Pass@1, achieving an increase of 13.87% on MBPP-sanitized (p-value=3.2e-05) and 16.83% on MBPP-ET (p-value=7.3e-05). Furthermore, when compared to the best-performing baselines (i.e., CoT or GPT-Engineer), ClarifyGPT (Human Feedback) also yields substantial improvements in Pass@1, showcasing enhancements of 9.53% on MBPP-sanitized (p-value=1.7e-04) and 9.52% on MBPP-ET (p-value=4.3e-04). This is mainly because ClarifyGPT can proficiently identify ambiguous requirements and raise targeted clarification questions. Users easily clarify their intentions by responding to these questions, thus facilitating the generation of more correct

Table 3. The Pass@1(%) of ClarifyGPT receiving human feedback and baselines on two benchmarks. Numbers in red denote ClarifyGPT (Human Feedback)'s relative improvements compared to the *Default*.

Methods	GPT-4		
	MBPP-sanitized	MBPP-ET	Average
Default	70.96	51.52	61.24
CoT	72.68	53.79	63.24
GPT-Engineer	73.77	54.96	64.37
ClarifyGPT (Human Feedback)	80.80	60.19	70.50
Relative Improvement	13.87% ↑	16.83% ↑	15.35% ↑

code by LLMs. It indicates that ClarifyGPT, as an interactive code generation framework, can support developers in writing code within real-world development contexts.

Answering RQ1: ClarifyGPT (Human Feedback) elevates the performance (Pass@1) of GPT-4 on MBPP-sanitized from 70.96% to 80.8%; elevates its performance on MBPP-ET from 51.52% to 60.19%. The relative improvement is 15.35% on average, outperforming the baselines.

5.2 RQ2: Performance with Simulated User Feedback

Setup. Due to the involvement of human participants, evaluating the interactive code generation framework ClarifyGPT is very expensive and hard to reproduce. A relatively simple solution is to conduct an offline evaluation [4]. However, it limits the system to selecting clarifying questions from a set of pre-defined or labeled questions, which does not transfer well to the practical development environment. In this RQ, we apply the User Simulation for Evaluation [16, 39] method to facilitate automated evaluations of ClarifyGPT across various LLMs and benchmarks, eliminating the necessity for direct user participation.

The most crucial aspect of simulating user feedback is to ensure that the created user feedback closely resembles the real feedback users would provide in the same environment. Low-fidelity simulations can result in ClarifyGPT receiving feedback that is challenging to encounter in actual practice, thereby yielding misleading outcomes and impacting our evaluation of ClarifyGPT's performance. Hence, we propose a high-fidelity user simulation method that leverages LLMs to generate user responses by providing LLMs with clarifying questions and ground-truth test cases. Our key insight is that the ground-truth test cases contain expected input-output examples, reflecting the desired functionality sought by users. Endowing LLMs with this prior knowledge facilitates their understanding of user intent and enables the generation of high-fidelity simulated user feedback. To instruct LLMs to solve this task, we design a prompt (as shown in Figure 2), which also consists of three parts: (1) an instruction, which describes the task (i.e., simulating the user responses) we want the LLMs to solve; (2) few-shot <requirement, ground-truth tests, clarifying questions, answers> quadruples as demonstrations, which help LLMs in understanding and solving the task; (3) a query, containing a user requirement and its ground-truth tests, which is fed to LLMs for generating simulated responses.

We apply three baselines (Section 4.5) and our ClarifyGPT to two SOTA LLMs (Section 4.2). We evaluate them on five benchmarks (Section 4.3) and compare their performance by calculating the Pass@1 metric (Section 4.4). To ensure a fair comparison, all baselines adopt the same experimental setup as our ClarifyGPT. Particularly, as GPT-Engineer also requires user feedback for code

Table 4. The Pass@1(%) of ClarifyGPT receiving simulated feedback and baselines on five benchmarks. Numbers in red denote ClarifyGPT (Simulated Feedback)'s relative improvements compared to the *Default*.

	Methods	HumanEval	HumanEval-ET	MBPP-sanitized	MBPP-ET	CoderEval	Average
ChatGPT	Default	64.63	57.32	65.57	46.68	37.42	54.32
	CoT	68.70	60.37	66.59	49.18	39.47	56.86
	GPT-Engineer	66.26	59.76	69.09	50.20	38.24	56.71
	ClarifyGPT (Simulated Feedback)	**74.39**	**64.84**	**74.08**	**55.58**	**42.94**	**62.37**
	Relative Improvement	15.10% ↑	13.12% ↑	12.98% ↑	19.07% ↑	14.75% ↑	15.00% ↑
GPT-4	Default	78.86	70.73	70.96	51.52	40.08	62.43
	CoT	80.10	72.56	72.68	53.79	42.13	64.25
	GPT-Engineer	79.27	71.75	73.77	54.96	41.10	64.17
	ClarifyGPT (Human Feedback)	\	\	**80.80**	**60.19**	\	**70.50**
	ClarifyGPT (Simulated Feedback)	**87.80**	**78.05**	**78.69**	**58.47**	**44.99**	**69.60**
	Relative Improvement	11.34% ↑	10.35% ↑	10.89% ↑	13.49% ↑	12.25% ↑	11.66% ↑

generation, we apply the same user simulation methods utilized by ClarifyGPT to facilitate GPT-Engineer in acquiring feedback.

Results. Table 4 presents the comparison results between the performance of ClarifyGPT receiving simulated feedback and other baselines in terms of code generation. The values in red are ClarifyGPT (Simulated Feedback)'s relative improvements compared to the *Default* baseline.

Overall, ClarifyGPT (Simulated Feedback) can significantly improve the performance of code generation, achieving gains across different LLMs and datasets (all the p-values are substantially smaller than 0.001). For GPT-4 model, compared with the *Default* baseline, ClarifyGPT (Simulated Feedback) demonstrates notable improvements in Pass@1 performance, achieving an increase of 11.34% on HumanEval, 10.35% on HumanEval-ET, 10.89% on MBPP-sanitized, 13.49% on MBPP-ET, and 12.25% on CoderEval. For ChatGPT model, when compared to the *Default* baseline, ClarifyGPT (Simulated Feedback) improves the performance of Pass@1 by 15.10%, 13.12%, 12.98%, 19.07%, and 14.75% on the five benchmarks, respectively. The results demonstrate that ClarifyGPT, which empowers LLMs to autonomously generate clarifying questions and refine user requirements based on user feedback, facilitates users in clarifying their intentions, thereby enhancing code generation performance by capturing user intentions.

We also note that, in comparison to the most related baseline (i.e., GPT-Engineer), ClarifyGPT (Simulated Feedback) exhibits superior performance with respect to the Pass@1 metric. Specifically, it achieves an average improvement of 11.45%, 8.65%, 6.95%, 8.56%, and 10.88% across the five benchmarks. We attribute the improvements to our novel techniques, i.e., ambiguous requirement identification and clarifying question generation. Posing clarifying questions for every user requirement results in needless LLM-Human interactions on unambiguous requirements, which places an additional burden on users and hurts the code generation performance when producing off-topic questions. While ClarifyGPT can effectively identify ambiguous requirements without any supervised training by conducting the code consistency check. The inconsistent code snippets are taken as input to help ClarifyGPT formulate targeted questions that guide users in clarifying ambiguity.

Besides, we observe that the performance of ClarifyGPT (Human Feedback) is slightly higher than that of ClarifyGPT (Simulated Feedback). This suggests that our user simulation method may generate user responses that do not fulfill the users' intentions. However, both methods can significantly improve the performance of code generation and achieve consistent gains across

Table 5. Experimental results of ClarifyGPT with different number of demonstrations. Numbers in red denote the relative improvement of ClarifyGPT compared to the *Default*.

	Methods	HumanEval	HumanEval-ET	MBPP-sanitized	MBPP-ET	CoderEval	Average
ChatGPT	Default	64.63	57.32	65.57	46.68	37.42	54.32
	ClarifyGPT (zero-shot)	65.85 1.9% ↑	58.13 1.4% ↑	67.07 2.3% ↑	48.01 2.8% ↑	39.26 4.9% ↑	55.66 2.7% ↑
	ClarifyGPT (one-shot)	72.80 12.6% ↑	60.98 6.4% ↑	70.96 8.2% ↑	51.52 10.4% ↑	40.70 8.8% ↑	59.39 9.3% ↑
	ClarifyGPT (two-shot)	73.92 14.4% ↑	63.21 10.3% ↑	72.60 10.7% ↑	53.63 14.9% ↑	42.13 12.6% ↑	61.10 12.6% ↑
	ClarifyGPT (three-shot)	**74.39** 15.1% ↑	**64.84** 13.1% ↑	**74.08** 13.0% ↑	**55.58** 19.1% ↑	**42.94** 14.8% ↑	**62.37** 15.0% ↑
GPT-4	Default	78.86	70.73	70.96	51.52	40.08	62.43
	ClarifyGPT (zero-shot)	79.26 0.5% ↑	70.73 0.0% -	72.13 1.6% ↑	52.22 1.4% ↑	41.10 2.5% ↑	63.09 1.2% ↑
	ClarifyGPT (one-shot)	83.93 6.4% ↑	72.76 2.9% ↑	75.88 6.9% ↑	55.97 8.6% ↑	41.92 4.6% ↑	66.09 5.9% ↑
	ClarifyGPT (two-shot)	85.15 8.0% ↑	75.61 6.9% ↑	77.75 9.6% ↑	56.67 10.0% ↑	43.56 8.7% ↑	67.75 8.6% ↑
	ClarifyGPT (three-shot)	**87.80** 11.3% ↑	**78.05** 10.3% ↑	**78.69** 10.9% ↑	**58.47**13.5% ↑	**44.99** 12.3% ↑	**69.60** 11.7% ↑

different LLMs and benchmarks, demonstrating the reliability of our simulation method's evaluation results.

Answering RQ2: ClarifyGPT (Simulated Feedback) improves the average performance (Pass@1) of GPT-4 across five benchmarks from 62.43% to 69.60%, improves the average performance of ChatGPT across five benchmarks from 54.32% to 62.37%. Their relative improvements are 11.66% and 15.00% respectively, and the average improvement is 13.33%.

5.3 RQ3: Performance for Different Number of Demonstrations

Setup. In this RQ, we investigate whether the increase or decrease in the number of demonstrations will affect the performance of ClarifyGPT on the code generation task. Specifically, due to the limitation of the input length of LLMs, we vary the number of demonstrations in the prompt from zero to three. Then, we apply the two LLMs to ClarifyGPT and its variants, and assess their performance of five benchmarks. We run these methods three times and report the average Pass@1 results as the final reports.

Results. Table 5 presents a comparison of the performance between ClarifyGPT and its variants. Overall, ClarifyGPT demonstrates robustness to the number of demonstrations in the prompts. When varying the number of demonstrations from zero to three, ClarifyGPT consistently outperforms the Default baseline across two LLMs and five benchmarks.

We can observe that, as expected, the performance of ClarifyGPT increases with the number of demonstrations. In particular, as the number of demonstrations in the prompt is incremented from zero to three, concerning ChatGPT, ClarifyGPT achieves an average performance increase from 55.66% to 62.37% across five benchmarks. For the GPT-4 model, ClarifyGPT's average performance increases from 63.09% to 69.60%. This is mainly because more demonstrations can provide a variety of situations and information to LLMs, enabling them to better comprehend the context of the problem and the required solution. Furthermore, LLMs can learn to generalize better through demonstrations, that is, to infer a solution to a new situation from a known demonstration. This allows LLMs to better adapt to different inputs and requirements.

We also find that ClarifyGPT's performance in the zero-shot setting exhibits a marginal improvement over the Default baseline, while its performance in the one-shot setting is significantly enhanced compared to that of the Default baseline. We attribute this difference to the fact that in the zero-shot setting, ClarifyGPT is expected to generate meaningful responses without any demonstrations, which can be particularly challenging for complex tasks (e.g., requiring LLMs to

generate targeted clarifying questions). What's more, zero-shot prompting relies solely on LLMs' pre-trained knowledge and the wording of the given prompts, which may not offer sufficient guidance or constraints for LLMs to produce accurate or contextually relevant responses. In contrast, the performance of ClarifyGPT with the one-shot setting is significantly higher than that in the zero-shot setting and is close to the performance of ClarifyGPT with the three-shot setting. This indicates that ClarifyGPT has strong generalization performance when only one demonstration is provided. We believe that in practical usage scenarios, utilizing ClarifyGPT in the one-shot setting can serve as a trade-off between effectiveness and efficiency.

Answering RQ3: Overall, ClarifyGPT demonstrates robustness to the number of demonstrations in the prompts. When varying the number of demonstrations from zero to three, ClarifyGPT consistently outperforms the Default baseline across two LLMs and five benchmarks.

6 HUMAN EVALUATION

In this section, we perform a human evaluation to further explore the reasons contributing to the enhancement of ClarifyGPT's performance.

6.1 Questions

We aims to address the following three questions:

(1) How effective is ClarifyGPT in detecting ambiguous requirements? ClarifyGPT distinguishes between ambiguous and unambiguous requirements, generating clarifying questions for the identified ambiguous ones. If ClarifyGPT fails to focus solely on questioning the ambiguous requirements, it may result in unnecessary interactions with users regarding unambiguous requirements, ultimately reducing efficiency and compromising the user experience. Therefore, to tackle this question, we invite ten participants to annotate the benchmark data as either ambiguous or unambiguous. Subsequently, we utilize these annotations as ground truth to assess the effectiveness of ClarifyGPT in detecting ambiguous requirements.

(2) How effective is ClarifyGPT in generating high-quality clarifying questions? The quality of clarifying questions also influences the efficiency and performance of code generation. High-quality clarifying questions can help users in expressing their intents clearly. Thus, in this question, we also invite the participants to assess whether the generated clarifying questions are high-quality from three aspects: relevance, comprehensiveness, and usefulness.

(3) How effective is ClarifyGPT in generating accurate code solutions for both ambiguous and unambiguous requirements? In Section 5, the experimental results demonstrate that ClarifyGPT can significantly enhance code generation performance. In this question, we aim to delve deeper into ClarifyGPT's effectiveness in generating accurate code solutions for ambiguous requirements and unambiguous requirements, respectively.

6.2 Performance in Detecting Ambiguous Requirements

Setup. Given the subjective nature of determining whether a requirement is ambiguous or unambiguous, and the labor-intensive process of manual annotation, we have limited our annotation efforts to the MBPP-sanitized benchmark, which consists of 427 problems. Specifically, we invite the ten participants involved in RQ1 to annotate the data. To ensure labeling accuracy, we employed a two-round annotation process: Initially, each participant independently assessed the category (ambiguous or unambiguous) of every problem within MBPP-sanitized. Subsequently, any conflicts in annotation were resolved through majority voting among the participants. The agreement

between the ten participants reached a Cohen's Kappa score of 0.86 The final annotation results are shown in Table 6.

Table 6. The results of manual annotation for MBPP-sanitized, alongside ClarifyGPT's performance in detecting ambiguous requirements, measured in terms of Precision, Recall, and F1-score metrics.

Methods	MBPP-sanitized		
Manual Annotations	Ambiguous 141	Unambiguous 286	Total 427
	Precision	Recall	F1-score
ClarifyGPT (ChatGPT)	72.35	87.23	79.10
ClarifyGPT (GPT-4)	88.57	87.94	88.25

Results. Table 6 demonstrates the performance of ClarifyGPT in detecting ambiguous requirements on MBPP-sanitized benchmark. Overall, ClarifyGPT demonstrates commendable effectiveness in this task. We can see that, for the ChatGPT model, ClarifyGPT achieves a precision of 72.35%, a recall of 87.23%, and an f1-score of 79.10%. Similarly, for the GPT-4 model, ClarifyGPT demonstrates precision, recall, and f1-score metrics of 88.57%, 87.94%, and 88.25%, respectively. We also note that ClarifyGPT performs better when using the GPT-4 model compared to when using the ChatGPT model. This observation is consistent with the results presented in Table 4, where ClarifyGPT (GPT-4) achieves a higher pass@1 metric than ClarifyGPT (ChatGPT).

6.3 Performance in Generating Clarifying Questions

Setup. In subsection 5.1 (RQ1), we gathered 140 identified ambiguous problems along with their corresponding clarifying questions. We recruited ten participants, assigned 42 problems to each participant, and asked them to answer the clarifying questions. After that, participants were prompted to evaluate whether they think the generated clarifying question are helpful. Specifically, we consider three metrics: (1) **Relevance**, determining the extent to which the generated clarifying questions are pertinent to the ambiguities; (2) **Comprehensiveness**, evaluating the adequacy of the generated clarifying questions in addressing all ambiguities; and (3) **Usefulness**, measuring the effectiveness of the generated clarifying questions in resolving ambiguities and clarifying intentions. Participants are tasked with rating the accuracy of detected ambiguous problems and assessing the relevance and comprehensiveness of the generated clarifying questions. Ratings for each metric ranged from 0 to 2, with 2 indicating a positive assessment, 1 indicating a neutral stance, and 0 indicating a negative assessment.

Table 7. The results of human evaluation. Numbers in parentheses denote the standard deviations.

Metrics	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
Relevance	1.83 (0.38)	1.90 (0.30)	1.74 (0.56)	1.86 (0.42)	1.88 (0.45)	1.79 (0.56)	1.93 (0.34)	1.76 (0.62)	1.81 (0.55)	1.95 (0.22)
Comprehensiveness	1.76 (0.53)	1.79 (0.52)	1.71 (0.64)	1.74 (0.59)	1.84 (0.47)	1.76 (0.58)	1.62 (0.76)	1.64 (0.66)	1.76 (0.53)	1.83 (0.49)
Usefulness	1.81 (0.45)	1.83 (0.49)	1.76 (0.62)	1.81 (0.50)	1.86 (0.47)	1.79 (0.56)	1.71 (0.64)	1.71 (0.60)	1.79 (0.56)	1.88 (0.40)

Results. The results of the human evaluation are shown in Table 7. The numbers in parentheses denote the standard deviations. Overall, the ten participants expressed positive evaluations regarding the quality of the generated clarification questions across all three assessment metrics. The average scores for relevance, comprehensiveness, and usefulness are 1.85, 1.75, and 1.80, respectively. Moreover, the standard deviations of all metrics are relatively small, indicating that their scores are about the same degree of concentration.

6.4 Performance in Generating Code for Ambiguous and Unambiguous Requirements

Setup. Drawing from the results of manual annotation for ambiguous and unambiguous requirements (illustrated in Table 6), we partition MBPP-sanitized and MBPP-ET [3] into "Ambiguous" and "Unambiguous" subsets. We then leverage the human feedback collected from RQ1 to calculate ClarifyGPT's performance, and compare the performance of ClarifyGPT receiving human feedback with that of the *Default* on both subsets of benchmarks.

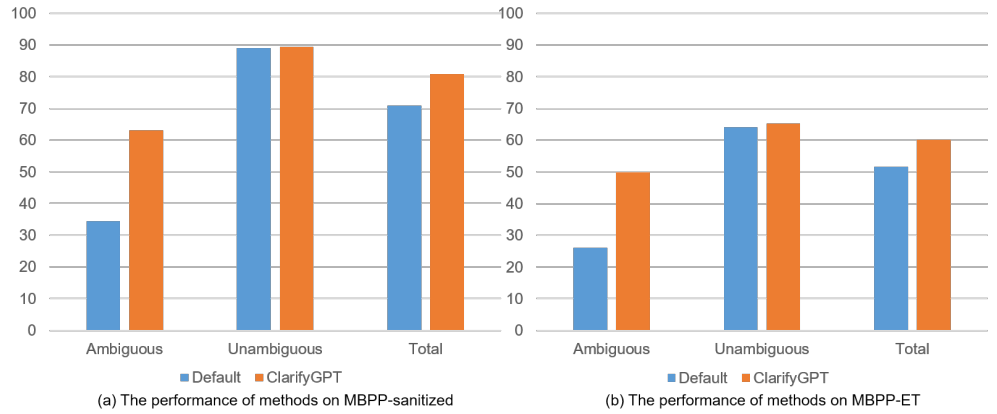

Fig. 3. The Pass@1(%) of ClarifyGPT receiving human feedback and *Default* on MBPP-sanitized and MBPP-ET benchmarks.

Results. The comparison results are presented in Figure 3. We observe that ClarifyGPT enhances code generation performance, particularly evident in the "Ambiguous" subsets of both benchmarks. Compared to the *Default*, ClarifyGPT achieves an increase of 13.87% on MBPP-sanitized and 16.83% on MBPP-ET. Notably, for the "Ambiguous" subset, ClarifyGPT outperforms the *Default* significantly, showcasing enhancements of 83.36% on MBPP-sanitized and 91.88% on MBPP-ET. For the "Unambiguous" subset, ClarifyGPT also demonstrates slight improvements. We attribute these improvements to ClarifyGPT's capability to effectively identify ambiguous requirements and generate high-quality clarifying questions to assist users in resolving ambiguities. Consequently, ClarifyGPT demonstrates significant enhancements over the *Default* approach, particularly within the "Ambiguous" subsets of benchmarks.

7 DISCUSSION

7.1 Benefits and Limitations

In this section, we discuss some potential benefits and limitations of our ClarifyGPT.

Benefits. (1) In contrast to prevailing LLM-based code generation methods [8, 23, 27] that leverage post-processing techniques to sample a substantial pool of candidate codes and then select one, ClarifyGPT aims to directly clarify the input requirements by asking clarifying questions. Hence, our framework contributes to the augmentation of interpretability in the code generated by LLMs. By clarifying specific details within the requirements or adding supplementary knowledge to them, users can readily discern corresponding alterations in the resulting code. This contributes to providing users with guidance on how to formulate requirements to improve code generation,

[3] As MBPP-ET shares identical requirements with MBPP-sanitized, their distributions of ambiguous and unambiguous requirements are also identical.

thereby facilitating a clearer understanding of the generated code. (2) Our ClarifyGPT improves the interactive skills of LLMs by empowering them with the ability to automatically ask clarifying questions for ambiguous requirements. In this way, it serves to facilitate users in identifying ambiguities within requirements and provides guidance in clarifying their intentions without requiring users to initially generate code and subsequently read and analyze code to refine requirements. Thus, ClarifyGPT enhances the user experience and production efficiency.

Limitations. (1) Ideally, our framework is applicable to all LLMs. However, ClarifyGPT necessitates that the LLMs possess a certain level of communicative competence, that is, the ability to comprehend human instructions and formulate clarifying questions. Thus, the LLMs applicable to our framework are limited, i.e., the LLMs without instruction tuning (e.g., InCoder [14] and CodeGen [33]) are not suitable as the base models applied to ClarifyGPT framework. (2) Due to the use of code consistency check to determine whether a requirement needs clarification, ClarifyGPT is required to generate test inputs for the requirement and compare the test outputs of the sampled solutions. Therefore, ClarifyGPT is not suitable for generating code with complex input (e.g., image or file). In addition, for some code that does not return output values (e.g., deep learning programs), using ClarifyGPT may also be subject to some limitations. (3) Utilizing ClarifyGPT can introduce additional overhead, primarily due to the code consistency check component, which entails sampling a specific quantity of programs (in this paper, we sample 25 programs for each problem). We conducted a comparative analysis of ClarifyGPT against baseline approaches in terms of cost per 100 problems. Specifically, upon utilizing the ChatGPT API, the average expenses incurred per 100 problems encountered were as follows: *Default* incurred an average cost of $0.017, CoT incurred an average cost of $0.058, GPT-Engineer incurred an average cost of $0.223, and ClarifyGPT incurred an average cost of $0.421. Note that ClarifyGPT offers performance improvements of approximately 10% over GPT-Engineer, at a cost less than twice that of GPT-Engineer. Furthermore, with advancements in LLM technology and decreasing costs for invoking LLM APIs, the cost disparity of ClarifyGPT is expected to diminish over time.

7.2 Threats to Validity

The first threat to validity is the potential for data leakage. Since these LLMs are trained on open-source code repositories, it is possible that some public benchmarks were included in their training data. This could bias our assessment of the proposed approach, as some model outputs may be influenced by prior exposure to these benchmarks. To mitigate this threat, we carefully select HumanEval [9], MBPP-sanitized [5], and their respective extended versions for our evaluation. HumanEval is a manually crafted problem-solving dataset, introduced by OpenAI for assessing Codex's performance. MBPP-sanitized, on the other hand, is a hand-verified subset of the MBPP dataset, comprising 427 Python problems that have undergone crowd-sourced verification. These datasets have undergone meticulous manual review and have been widely employed in previous research studies [8, 23, 46].

The second threat to validity is the user simulation for evaluation. Due to the involvement of human participants, evaluating ClarifyGPT, an interactive code generation framework, is very expensive and hard to reproduce. Thus, we propose a user simulation method to facilitate automated evaluations of ClarifyGPT across various LLMs and benchmarks. However, low-fidelity simulations can result in ClarifyGPT receiving feedback that is challenging to encounter in actual practice, thereby yielding misleading outcomes and impacting our evaluation of ClarifyGPT's performance. To mitigate this threat, we design a special prompt to provide LLMs with clarifying questions and ground-truth test cases. By endowing LLMs with this prior knowledge, ClarifyGPT facilitates LLMs' understanding of user intent and enables the generation of high-fidelity simulated user feedback. The results show that the performance of ClarifyGPT (Simulated Feedback) is very

close to that of ClarifyGPT (Human Feedback), proving that our proposed simulation method can serve as a good proxy for the automatic evaluation of ClarifyGPT, eliminating the necessity for direct user participation.

The third threat pertains to the generalizability of our experimental results. To mitigate this threat, on one hand, we have taken care to select two representative chat LLMs (ChatGPT and GPT-4) as our base models and five widely-used datasets as the evaluation subjects. We apply the two LLMs to our ClarifyGPT and assess their performance on these five datasets. On the other hand, considering the inherent sensitivity of LLMs to prompts, we run baselines and ClarifyGPT three times and report the average results as the final results. Despite the efforts described, doubts remain about whether ClarifyGPT can generalize beyond the experimental datasets, as these datasets may have been part of the training sets for LLMs. We recommend that developers conduct performance tests before integrating ClarifyGPT into their projects. In future work, we will delve deeper into ClarifyGPT's performance in real-world development scenarios.

8 CONCLUSION

In this paper, motivated by the observation that human developers typically ask clarifying questions when they are faced with ambiguous requirements, we argue that empowering LLMs with the ability to automatically clarify ambiguous requirements can improve code generation. To this end, we propose ClarifyGPT, a code generation framework that enables LLMs to identify ambiguous requirements and generate targeted clarifying questions. Specifically, ClarifyGPT consists of four main stages, i.e., test input generation, code consistency check, reasoning-based question generation, and enhanced code generation. For a given requirement, ClarifyGPT first generates high-quality test inputs by using prompting techniques and heuristic mutations. Then, it utilizes the generated test inputs to conduct a consistency evaluation and identify the ambiguous requirements. Next, ClarifyGPT formulates targeted clarifying questions for the identified ambiguous requirements by prompting LLMs to engage in intermediate reasoning. Finally, it incorporates the clarifying questions and their feedback to refine the original requirement and generate the final code solution based on the refined prompt. In the evaluation part, we first apply GPT-4 to ClarifyGPT and recruit ten participants to evaluate its performance on two public benchmarks. The human evaluation results show that ClarifyGPT achieves a relative improvement of up to 16.83% in Pass@1 compared to the *Default* baseline. Additionally, to automate the evaluation of ClarifyGPT, we introduce a high-fidelity simulation method to simulate user feedback. We conduct comprehensive experiments on five benchmarks (i.e., HumanEval, HumanEval-ET, MBPP-sanitized, MBPP-ET, and CoderEval) using two LLMs (i.e., GPT-4 and ChatGPT). The extensive results illustrate that ClarifyGPT improves the average performance of GPT-4 across five benchmarks from 62.43% to 69.60%, and improves the average performance of ChatGPT across five benchmarks from 54.32% to 62.37%. In future work, we intend to integrate automatic prompt optimization techniques into the framework to automatically generate and improve prompts for various tasks or datasets. Furthermore, we plan to explore ClarifyGPT's efficacy in real-world development scenarios.

ACKNOWLEDGMENTS

We sincerely appreciate anonymous reviewers for their constructive and insightful suggestions for improving this manuscript. This work was supported by the National Natural Science Foundation of China Grant No. 62332001, No. 62232016, No. 62072442, and No. 62272445, Youth Innovation Promotion Association Chinese Academy of Sciences, Basic Research Program of ISCAS Grant No. ISCAS-JCZD-202304, and Major Program of ISCAS Grant No. ISCAS-ZD-202302, and the grant from Huawei.

REFERENCES

[1] 2023. CoderEval. https://github.com/CoderEval/CoderEval.

[2] 2023. Website. https://github.com/ClarifyGPT/ClarifyGPT.

[3] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2021, Online, June 6-11, 2021*. 2655–2668. https://doi.org/10.18653/v1/2021.naacl-main.211

[4] Mohammad Aliannejadi, Hamed Zamani, Fabio Crestani, and W. Bruce Croft. 2019. Asking Clarifying Questions in Open-Domain Information-Seeking Conversations. In *Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2019, Paris, France, July 21-25, 2019*, Benjamin Piwowarski, Max Chevalier, Éric Gaussier, Yoelle Maarek, Jian-Yun Nie, and Falk Scholer (Eds.). ACM, 475–484. https://doi.org/10.1145/3331184.3331265

[5] Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. *CoRR* abs/2108.07732 (2021). arXiv:2108.07732 https://arxiv.org/abs/2108.07732

[6] Sid Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, Michael Pieler, USVSN Sai Prashanth, Shivanshu Purohit, Laria Reynolds, Jonathan Tow, Ben Wang, and Samuel Weinbach. 2022. GPT-NeoX-20B: An Open-Source Autoregressive Language Model. *CoRR* abs/2204.06745 (2022). https://doi.org/10.48550/arXiv.2204.06745 arXiv:2204.06745

[7] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott M. Lundberg, Harsha Nori, Hamid Palangi, Marco Túlio Ribeiro, and Yi Zhang. 2023. Sparks of Artificial General Intelligence: Early experiments with GPT-4. *CoRR* abs/2303.12712 (2023). https://doi.org/10.48550/ARXIV.2303.12712 arXiv:2303.12712

[8] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. CodeT: Code Generation with Generated Tests. *CoRR* abs/2207.10397 (2022). https://doi.org/10.48550/arXiv.2207.10397 arXiv:2207.10397

[9] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021). arXiv:2107.03374 https://arxiv.org/abs/2107.03374

[10] Kaustubh D. Dhole. 2020. Resolving Intent Ambiguities by Retrieving Discriminative Clarifying Questions. *CoRR* abs/2008.07559 (2020). arXiv:2008.07559 https://arxiv.org/abs/2008.07559

[11] Yihong Dong, Jiazheng Ding, Xue Jiang, Zhuo Li, Ge Li, and Zhi Jin. 2023. CodeScore: Evaluating Code Generation by Learning Code Execution. *CoRR* abs/2301.09043 (2023). https://doi.org/10.48550/arXiv.2301.09043 arXiv:2301.09043

[12] Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. 2023. Self-collaboration Code Generation via ChatGPT. *CoRR* abs/2304.07590 (2023). https://doi.org/10.48550/arXiv.2304.07590 arXiv:2304.07590

[13] Zachary Eberhart and Collin McMillan. 2022. Generating Clarifying Questions for Query Refinement in Source Code Search. In *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2022, Honolulu, HI, USA, March 15-18, 2022*. IEEE, 140–151. https://doi.org/10.1109/SANER53432.2022.00028

[14] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. InCoder: A Generative Model for Code Infilling and Synthesis. *CoRR* abs/2204.05999 (2022). https://doi.org/10.48550/arXiv.2204.05999 arXiv:2204.05999

[15] Shuzheng Gao, Xin-Cheng Wen, Cuiyun Gao, Wenxuan Wang, and Michael R. Lyu. 2023. Constructing Effective In-Context Demonstration for Code Intelligence Tasks: An Empirical Study. *CoRR* abs/2304.07575 (2023). https://doi.org/10.48550/ARXIV.2304.07575 arXiv:2304.07575

[16] Michael D Gordon. 1990. Evaluating the effectiveness of information retrieval systems using simulated queries. *Journal of the American Society for Information Science* 41, 5 (1990), 313–323.

[17] Xue Jiang, Yihong Dong, Lecheng Wang, Qiwei Shang, and Ge Li. 2023. Self-planning Code Generation with Large Language Model. *CoRR* abs/2303.06689 (2023). https://doi.org/10.48550/arXiv.2303.06689 arXiv:2303.06689

[18] Kimiya Keyvan and Jimmy Xiangji Huang. 2023. How to Approach Ambiguous Queries in Conversational Search: A Survey of Techniques, Approaches, Tools, and Challenges. *ACM Comput. Surv.* 55, 6 (2023), 129:1–129:40. https:

//doi.org/10.1145/3534965

[19] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large Language Models are Zero-Shot Reasoners. *CoRR* abs/2205.11916 (2022). https://doi.org/10.48550/arXiv.2205.11916 arXiv:2205.11916

[20] Dmitrii Krasheninnikov, Egor Krasheninnikov, and David Krueger. 2022. Assistance with large language models. In *NeurIPS ML Safety Workshop*.

[21] Lorenz Kuhn, Yarin Gal, and Sebastian Farquhar. 2023. CLAM: Selective Clarification for Ambiguous Questions with Generative Language Models. (2023).

[22] Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy Liang. 2019. SPoC: Search-based Pseudocode to Code. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*. 11883–11894. https://proceedings.neurips.cc/paper/2019/hash/7298332f04ac004a0ca44cc69ecf6f6b-Abstract.html

[23] Shuvendu K. Lahiri, Aaditya Naik, Georgios Sakkas, Piali Choudhury, Curtis von Veh, Madanlal Musuvathi, Jeevana Priya Inala, Chenglong Wang, and Jianfeng Gao. 2022. Interactive Code Generation via Test-Driven User-Intent Formalization. *CoRR* abs/2208.05950 (2022). https://doi.org/10.48550/arXiv.2208.05950 arXiv:2208.05950

[24] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. 2023. CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 919–931. https://doi.org/10.1109/ICSE48619.2023.00085

[25] Haau-Sing Li, Mohsen Mesgar, André F. T. Martins, and Iryna Gurevych. 2023. Python Code Generation by Asking Clarification Questions. 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*. Association for Computational Linguistics, 14287–14306. https://doi.org/10.18653/v1/2023.acl-long.799

[26] Jia Li, Ge Li, Yongmin Li, and Zhi Jin. 2023. Enabling Programming Thinking in Large Language Models Toward Code Generation. *CoRR* abs/2305.06599 (2023). https://doi.org/10.48550/ARXIV.2305.06599 arXiv:2305.06599

[27] Yujia Li, David H. Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-Level Code Generation with AlphaCode. *CoRR* abs/2203.07814 (2022). https://doi.org/10.48550/arXiv.2203.07814 arXiv:2203.07814

[28] Chao Liu, Xuanlin Bao, Hongyu Zhang, Neng Zhang, Haibo Hu, Xiaohong Zhang, and Meng Yan. 2023. Improving ChatGPT Prompt for Code Generation. *CoRR* abs/2305.08360 (2023). https://doi.org/10.48550/ARXIV.2305.08360 arXiv:2305.08360

[29] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. *CoRR* abs/2305.01210 (2023). https://doi.org/10.48550/arXiv.2305.01210 arXiv:2305.01210

[30] Sewon Min, Julian Michael, Hannaneh Hajishirzi, and Luke Zettlemoyer. 2020. AmbigQA: Answering Ambiguous Open-domain Questions. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP 2020, Online, November 16-20, 2020*, Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, 5783–5797. https://doi.org/10.18653/V1/2020.EMNLP-MAIN.466

[31] Noor Nashid, Mifta Sintaha, and Ali Mesbah. 2023. 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*. IEEE, 2450–2462. https://doi.org/10.1109/ICSE48619.2023.00205

[32] Feng Nie, Meixi Chen, Zhirui Zhang, and Xu Cheng. 2022. Improving Few-Shot Performance of Language Models via Nearest Neighbor Calibration. *CoRR* abs/2212.02216 (2022). https://doi.org/10.48550/ARXIV.2212.02216 arXiv:2212.02216

[33] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. https://openreview.net/pdf?id=iaYcJKpY2B_

[34] OpenAI. 2022. ChatGPT. https://openai.com/blog/chatgpt/.

[35] OpenAI. 2023. GPT-4 Technical Report. *CoRR* abs/2303.08774 (2023). https://doi.org/10.48550/arXiv.2303.08774 arXiv:2303.08774

[36] Anton Osika. 2023. GPT-Engineer. https://github.com/AntonOsika/gpt-engineer/.

[37] Sudha Rao and Hal Daumé III. 2019. Answer-based Adversarial Training for Generating Clarification Questions. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short*

Papers). Association for Computational Linguistics, 143–155. https://doi.org/10.18653/v1/n19-1013

[38] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. Adaptive Test Generation Using a Large Language Model. *CoRR* abs/2302.06527 (2023). https://doi.org/10.48550/ARXIV.2302.06527 arXiv:2302.06527

[39] Ivan Sekulic, Mohammad Aliannejadi, and Fabio Crestani. 2022. Evaluating Mixed-initiative Conversational Search Systems via User Simulation. In *WSDM '22: The Fifteenth ACM International Conference on Web Search and Data Mining, Virtual Event / Tempe, AZ, USA, February 21 - 25, 2022*. ACM, 888–896. https://doi.org/10.1145/3488560.3498440

[40] Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. 2022. Repository-Level Prompt Generation for Large Language Models of Code. *CoRR* abs/2206.12839 (2022). https://doi.org/10.48550/arXiv.2206.12839 arXiv:2206.12839

[41] Jan Trienes and Krisztian Balog. 2019. Identifying Unclear Questions in Community Question Answering Websites. In *Advances in Information Retrieval - 41st European Conference on IR Research, ECIR 2019, Cologne, Germany, April 14-18, 2019, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 11437)*. Springer, 276–289. https://doi.org/10.1007/978-3-030-15712-8_18

[42] Vasudev Vikram, Caroline Lemieux, and Rohan Padhye. 2023. Can Large Language Models Write Good Property-Based Tests? *CoRR* abs/2307.04346 (2023). https://doi.org/10.48550/ARXIV.2307.04346 arXiv:2307.04346

[43] Jian Wang and Wenjie Li. 2021. Template-guided Clarifying Question Generation for Web Search Clarification. In *CIKM '21: The 30th ACM International Conference on Information and Knowledge Management, Virtual Event, Queensland, Australia, November 1 - 5, 2021*. ACM, 3468–3472. https://doi.org/10.1145/3459637.3482199

[44] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-Consistency Improves Chain of Thought Reasoning in Language Models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. https://openreview.net/pdf?id=1PL1NIMMrw

[45] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*. Association for Computational Linguistics, 8696–8708. https://doi.org/10.18653/v1/2021.emnlp-main.685

[46] Zhiruo Wang, Shuyan Zhou, Daniel Fried, and Graham Neubig. 2022. Execution-Based Evaluation for Open-Domain Code Generation. *CoRR* abs/2212.10481 (2022). https://doi.org/10.48550/arXiv.2212.10481 arXiv:2212.10481

[47] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed H. Chi, Quoc Le, and Denny Zhou. 2022. Chain of Thought Prompting Elicits Reasoning in Large Language Models. *CoRR* abs/2201.11903 (2022). arXiv:2201.11903 https://arxiv.org/abs/2201.11903

[48] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *MAPS@PLDI 2022: 6th ACM SIGPLAN International Symposium on Machine Programming, San Diego, CA, USA, 13 June 2022*. ACM, 1–10. https://doi.org/10.1145/3520312.3534862

[49] Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. 2024. CoderEval: A Benchmark of Pragmatic Code Generation with Generative Pre-trained Models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 37:1–37:12. https://doi.org/10.1145/3597503.3623316

[50] Michal Zalewski. 2018. American fuzzing lop. https://lcamtuf.coredump.cx/afl/.

[51] Lingming Zhang, Darko Marinov, Lu Zhang, and Sarfraz Khurshid. 2011. An Empirical Study of JUnit Test-Suite Reduction. In *IEEE 22nd International Symposium on Software Reliability Engineering, ISSRE 2011, Hiroshima, Japan, November 29 - December 2, 2011*, Tadashi Dohi and Bojan Cukic (Eds.). IEEE Computer Society, 170–179. https://doi.org/10.1109/ISSRE.2011.26

[52] Tianyi Zhang, Tao Yu, Tatsunori B. Hashimoto, Mike Lewis, Wen-tau Yih, Daniel Fried, and Sida I. Wang. 2022. Coder Reviewer Reranking for Code Generation. *CoRR* abs/2211.16490 (2022). https://doi.org/10.48550/arXiv.2211.16490 arXiv:2211.16490

Received 2023-09-28; accepted 2024-04-16