ChatCoder: Human-in-loop Refine Requirement Improves LLMs' Code Generation

Anonymous ACL submission

Abstract

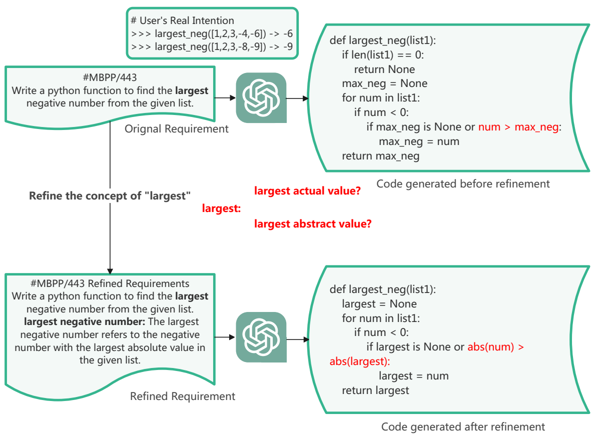
Large language models have shown good performances in generating code to meet human requirements. However, human requirements expressed in natural languages can be vague, incomplete, and ambiguous, leading large language models to misunderstand human requirements and make mistakes. Worse, it is difficult for a human user to refine the requirement. To help human users refine their requirements and improve large language models' code generation performances, we propose ChatCoder, a method to refine the requirements via chatting with large language models. We design a chat scheme in which the large language models will guide the human users to refine their expression of requirements to be more precise, unambiguous, and complete than before. Experiments show that ChatCoder has improved by a large margin. Besides, ChatCoder has the advantage over refine-based methods and LLMs fine-tuned via human response.

1 Introduction

Code generation is to generate source code that satisfies the human user's requirements expressed in natural language. Recently large language models have shown impressive abilities in code generation. For example, OpenAI's GPT-4(OpenAI, 2023) passes 67% of the tests of HumanEval(Chen et al., 2021), and so are the open-source LLMs, e.g., (Gunasekar et al., 2023) passes 50.6% of the tests. It's promising to deploy the LLM as an assistant for humans in real-world software production workflow.

Although promising, the LLMs struggle with poor requirement expressions. In practice, requirements are written in natural language. The natural language expressions can be vague, incomplete and ambiguous, misleading an LLM to generate the wrong code. Figure 1 shows a real-world example from a widely-used code generation benchmark, MBPP. The original expression of the requirement is to find 'the largest negative number' without explaining the definition of 'the largest negative number'. The real intent of the requirement is to find the negative number with the largest **absolute** value. However, according to the generated code from gpt-3.5-turbo, the LLM is misled to look for the negative number with the largest **actual** value. The ambiguity of 'the largest negative number' leads to incorrect programs from LLMs.

The above problem can be solved via *requirements refinement*, a widely used technique in real development scenarios. **Requirements refinement is the process of revealing the underlying dependencies and hidden structures**(Liu, 2008). It alleviates potential ambiguities by revealing more requirement details. As shown in Figure 1, we reveal the hidden structure of 'the largest' as 'the largest absolute value' to the LLM and get the correctly generated code.



```
# User's Real Intention
>>> largest_neg([1,2,3,-4,-6]) -> -6
>>> largest_neg([1,2,3,-8,-9]) -> -9
```

#MBPP/443
Write a python function to find the **largest** negative number from the given list.

```
def largest_neg(list1):
    if len(list1) == 0:
        return None
    max_neg = None
    for num in list1:
        if num < 0:
            if max_neg is None or num > max_neg:
                max_neg = num
    return max_neg
```

#MBPP/443 Refined Requirements
Write a python function to find the **largest** negative number from the given list.
largest negative number: The largest negative number refers to the negative number with the largest absolute value in the given list.

```
def largest_neg(list1):
    largest = None
    for num in list1:
        if num < 0:
            if largest is None or abs(num) >
abs(largest):
                largest = num
    return largest
```

Orignal Requirement
Code generated before refinement
Refine the concept of "largest"
largest:
largest actual value?
largest abstract value?
Refined Requirement
Code generated after refinement

Figure 1: Example of Refinement Improving Code Generation Performance

Inspired by the above observation, we introduce requirement refinement into code generation and propose ChatCoder, a new code generation approach. ChatCoder utilises the human-in-the-loop mechanism to refine requirements, asking the LLM and the human user to chat to extend the original requirement expression. By the cooperation of refinement, the details are settled, the disambiguation is clarified, and the more informative and precise refined requirement expressions are eventually formed. Then, the LLM generates code based on the refined version of the requirement. Compared with the previous code generation approaches, ChatCoder mitigates the disambiguation in the requirement expression, improving the LLM's code generation accuracy.

A key challenge in ChatCoder is to design the human-in-the-loop mechanism. As a code generation stakeholder, the human user has to participate in the refinement (iee, 1998). ChatCoder should provide an effective way of communication between an LLM and a human. We propose a two-round dialogue scheme for the human-in-the-loop requirement refinement. In each round, the LLM extends the requirement expressions obtained from the previous round following our designed requirement extension framework. Then, the human user checks the extended expressions and edits the mistakes. The edited expressions are prepared for the next round of extension or output as the finalised refined version.

We apply ChatCoder to 2 popular LLMs, gpt-3.5-turbo and gpt-4. Our method is evaluated on two popular benchmarks, Sanitized-MBPP and HumanEval. The test results show that the refined requirements with ChatCoder improve the LLM's code generation performances by a large margin, at an average percentage of 10.

Our contribution is summarised as follows:

1): We find and raise the problem that human's poor requirement expressions in natural language limit LLMs' ability to generate better programs.

2): We propose ChatCoder, the first approach using human-in-loop requirement refinement to enhance code generation.

3): We evaluate ChatCoder on Sanitized-MBPP and Humaneval benchmarks. The extensive experiments show that ChatCoder effectively enhances the LLMs' code generation performances.

2 background

2.1 Large Language Model for Code Generation

Large language models are currently pre-trained Transformer-based language models with at least tens of billions of parameters. The first well-known large language model is OpenAI's GPT-3 (Brown et al., 2020) which has extraordinary code generation ability. Following GPT-3, a series of business-oriented close-source large language models have been proposed, e.g. GPT-3.5 and GPT-4, and several open-source large language models for code-related tasks have been published, e.g., StarCoder(Li et al., 2023b), CodeT5+(Wang et al., 2023). WizardCoder(Luo et al., 2023) etc. They all behave well in code generation.

The prompting technique is the current way of using LLM to generate code. A prompt is a piece of carefully designed text, e.g., the instructions for the LLM. The prompt will be the LLM's input, influencing the LLM to generate what the human user expects. In code generation, researchers have explored multiple prompting techniques in order for the LLM to generate code satisfying the human users' requirements more precisely. For example, Li et al. (2023a) proposes to provide examples closely related to the programming tasks in the prompt as examples for the LLM. Jiang et al. (2023) proposes that decomposing the programming task in the prompt helps large language models solve complex problems. The experiments prove that the carefully designed prompts improve the LLMs' code generation performances.

2.2 Requirements Refinement

Requirements refinement is the process of revealing the underlying dependencies and hidden structures, serving as a start from requirement to design. Requirements refinement is important because many users in practice do not understand what functions they want precisely at the beginning of a software project(Liu, 2002), leading to poor original requirement expressions. Thus, the users and software suppliers need to refine the requirements to sort out the users' true needs.

Previous studies of requirements refinement focus on providing a formal method for the software supplier to analyse and refine the software customer's requirements. Liu (2008) raises a hierarchical framework from the business level to the component level to refactor the customer's requirements. de Jong et al. (2000) propose formal refinement patterns for goal-driven requirements elaboration via KAOS. Liu (2002) proposes to use the SOFL language to describe the refinement process and raises the model of successive refinements in which the requirements refinement is a process from coarse to fine with a loop back.

Requirements refinement requires collaboration with both the software provider and the software user. On the one hand, the refinement methods are frameworks that require domain knowledge and an understanding of actual user requirements. On the other hand, the users may not understand the software design enough, so their expressions may not correctly express their requirements. **So the key to requirement refinement techniques is to design an efficient way of cooperation for the user and the software to find the necessary points to extend and reach an agreement on the requirement.**

3 Methodology

3.1 Overall Structure of ChatCoder

ChatCoder is a code generation method. It improves LLM's code generation ability by refining requirements via the chat between LLMs and humans. ChatCoder comprises a dialogue schema following which the LLM and the human can communicate to refine requirements effectively and efficiently, resulting in an accurate and informative requirement expression. Based on the refined requirement expression, the LLM can generate code fulfilling the human's need precisely.

Specifically, ChatCoder refines the original requirement expressions via a two-round dialogue, illustrated in Fig 2. The first round is *Paraphrase and Extend*. Following our designed command, the LLM paraphrase the original requirement expression and extends it in six specific angles, necessary for the precise map from human needs in NL to to the corresponding code. Then the human programmer reviews the LLM's expanded expression and edits the mistakes and missing parts. The second round is *Going-deep and Loop-back*. Following our designed command, the LLM asks the human programmer about its further confusion of the final edited requirement expressions in the first round. If a question is reasonable to the human programmer, it is answered directly. Otherwise, the human programmer needs to review the expression in the first step to find and correct the mistake leading to the unreasonable question. The final refined requirement is the combination of the corrected expression in the first step and the reasonable questions in the second round and their corresponding answers.

In the following paragraphs, we will explain the design of each round in detail.

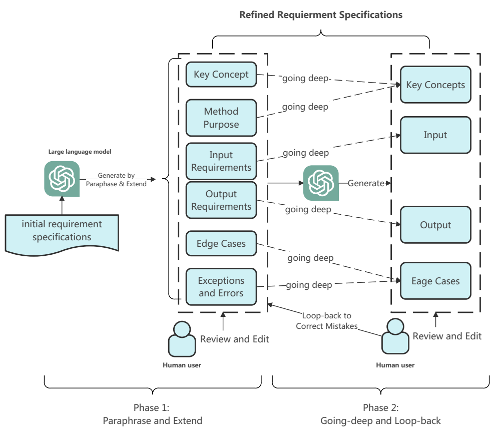


Figure 2: Overall Structure of the ChatCoder Dialogue Framework

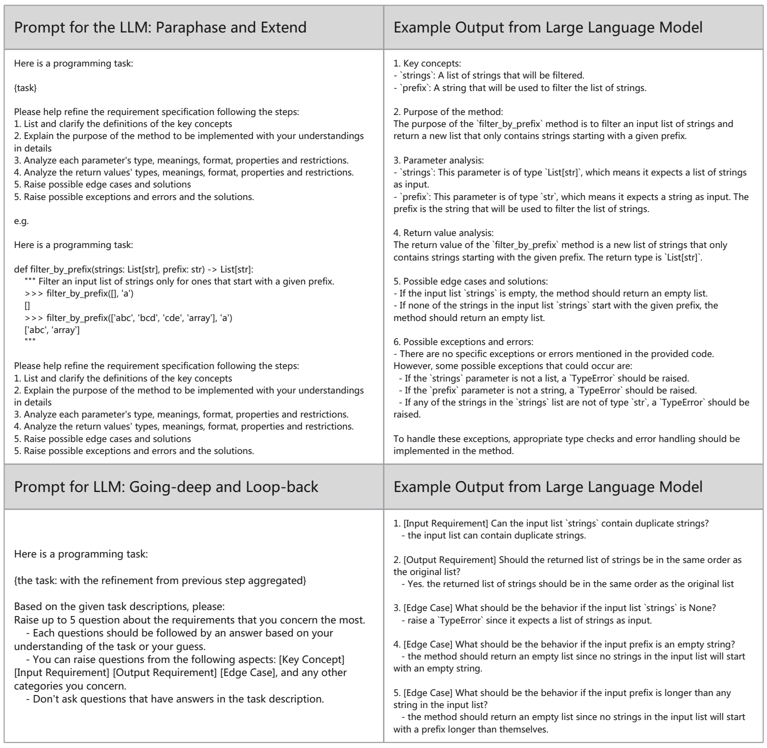
3.2 Paraphrase and Extend

The *Paraphrase and Extend* round is to extend the original vague, incomplete and ambitious requirement expression into clear and accurate requirement statements for coding, via the chat between LLM and the human programmer. In concrete, first the LLM paraphrases and extends the original requirement expression from six angles important for correct programming. Then the human programmer reviews the extended requirement expression and edits the wrong and the missing.

ChatCoder uses a hand-craft prompt to command the LLM to paraphrase and extend the expressions, illustrated in Figure 3. The prompt contains: 1) the instruction to paraphrase and extend; 2): the angles for extending the requirement. The design of the angles is important since the angles are the scaffold for both the LLM and the human programmers to sort out what the requirement is. Inspired by *IEEE Recommended Practice for Software Requirements Specifications*, we propose six angles from which the programmers are likely to ask for clarification of the requirement for correct programming. They are presented as follows:

Key Concepts The LLM is required to extract and explain the key concepts involved in the user's requirements, including objects and actions. By extending this angle, the user and the large language model can align their understanding of the key concepts, setting a firm basis for further discussion.

Method Purpose The LLM is required to paraphrase the design of how to fulfil the proposed requirements in detail. The LLM's design reflects its ongoing implementation based on the LLM's understanding of the initial requirement expression



Prompt for the LLM: Paraphase and Extend

```
Here is a programming task:

{task}

Please help refine the requirement specification following the steps:
1. List and clarify the definitions of the key concepts
2. Explain the purpose of the method to be implemented with your understandings
in details
3. Analyze each parameter's type, meanings, format, properties and restrictions.
4. Analyze the return values' types, meanings, format, properties and restrictions.
5. Raise possible edge cases and solutions
5. Raise possible exceptions and errors and the solutions.

e.g.

Here is a programming task:

def filter_by_prefix(strings: List[str], prefix: str) -> List[str]:
   """ Filter an input list of strings only for ones that start with a given prefix.
   >>> filter_by_prefix([], 'a')
   []
   >>> filter_by_prefix(['abc', 'bcd', 'cde', 'array'], 'a')
   ['abc', 'array']
   """

Please help refine the requirement specification following the steps:
1. List and clarify the definitions of the key concepts
2. Explain the purpose of the method to be implemented with your understandings
in details
3. Analyze each parameter's type, meanings, format, properties and restrictions.
4. Analyze the return values' types, meanings, format, properties and restrictions.
5. Raise possible edge cases and solutions
5. Raise possible exceptions and errors and the solutions.
```

Example Output from Large Language Model

```
1. Key concepts:
- `strings`: A list of strings that will be filtered.
- `prefix`: A string that will be used to filter the list of strings.

2. Purpose of the method:
The purpose of the `filter_by_prefix` method is to filter an input list of strings and
return a new list that only contains strings starting with a given prefix.

3. Parameter analysis:
- `strings`: This parameter is of type `List[str]`, which means it expects a list of strings
as input.
- `prefix`: This parameter is of type `str`, which means it expects a string as input. The
prefix is the string that will be used to filter the list of strings.

4. Return value analysis:
The return value of the `filter_by_prefix` method is a new list of strings that only
contains strings starting with the given prefix. The return type is `List[str]`.

5. Possible edge cases and solutions:
- If the input list `strings` is empty, the method should return an empty list.
- If none of the strings in the input list `strings` start with the given prefix, the
method should return an empty list.

6. Possible exceptions and errors:
- There are no specific exceptions or errors mentioned in the provided code.
However, some possible exceptions that could occur are:
  - If the `strings` parameter is not a list, a `TypeError` should be raised.
  - If the `prefix` parameter is not a string, a `TypeError` should be raised.
  - If any of the strings in the `strings` list are not of type `str`, a `TypeError` should be
raised.

To handle these exceptions, appropriate type checks and error handling should be
implemented in the method.
```

Prompt for LLM: Going-deep and Loop-back

```
Here is a programming task:

{the task: with the refinement from previous step aggregated}

Based on the given task descriptions, please:
Raise up to 5 question about the requirements that you concern the most.
   - Each questions should be followed by an answer based on your
understanding of the task or your guess.
   - You can raise questions from the following aspects: [Key Concept]
[Input Requirement] [Output Requirement] [Edge Case], and any other
categories you concern.
   - Don't ask questions that have answers in the task description.
```

Example Output from Large Language Model

```
1. [Input Requirement] Can the input list `strings` contain duplicate strings?
  - the input list can contain duplicate strings.

2. [Output Requirement] Should the returned list of strings be in the same order as
the original list?
  - Yes. the returned list of strings should be in the same order as the original list

3. [Edge Case] What should be the behavior if the input list `strings` is None?
  - raise a `TypeError` since it expects a list of strings as input.

4. [Edge Case] What should be the behavior if the input prefix is an empty string?
  - the method should return an empty list since no strings in the input list will start
with an empty string.

5. [Edge Case] What should be the behavior if the input prefix is longer than any
string in the input list?
  - the method should return an empty list since no strings in the input list will start
with a prefix longer than themselves.
```

Figure 3: Prompts for Large Language Models and Example Outputs

and its inference for the incomplete expression, revealing the error and incompleteness of the requirement expression.

Input Requirements The LLM is required to extend the requirements for the method's inputs, including the parameters' types, actual meaning, boundaries and properties. The type, boundary and property are important in programming by easily missing. And the human can check whether the LLM correctly understands the parameters' meanings.

Output Requirements The LLM is required to extend the requirements for the method outputs, including the types, the meaning and the format. Like the input requirements, the output requirements are easily missing and misunderstood, making it necessary to check them.

Edge Cases The LLM is required to extend possible edge cases and solutions. Since a method can run in complicated outer environments, the input and the global variable states may not fulfil the method's running preconditions. So properly handling edge cases is necessary for a robust method implementation but can be easily ignored by software customers.

Exceptions and Errors The LLM is required to extend the solutions for possible exceptions and errors during the method's execution. Like edge cases, handling exceptions and errors are necessary but can be easily missed by the users because of their unprofessional software design. The large language model must be analysed, raise solutions and wait for the users' review.

When the LLM finishes paraphrasing and extending the original expression, the coarsely refined requirements are sent to the human stakeholder to review item by item. The human stakeholder, i.e., the programmer who submits the programming requirements to the LLM, should check whether the refined requirements fulfil the needs, correcting the mistakes generated by the LLM.

3.3 Going-deep and Loop-back

In this round, the large language model is asked to going-deep: to further refine the requirements based on the specifications obtained in *Paraphrase and Extend*; the human user is requested to loop back: check for possible errors in the reviews and the errors corrected.

Going-deep The large language model is asked to raise questions in the angles based on the existing specifications obtained in *Paraphrase and Extend*. The instruction for the large language model is also wrapped in a prompt, presented in Figure 3. In this round, we let the LLM ask questions in a free form for what confused the most about the requirements, then give possible answers based on its observations or assumptions. We design *Going-deep* to refine the requirements further because the large language model is a black box, and it is hard to say we have used up its potential to refine requirements through *Paraphrase and Extend*.

Loop-back The user is asked to review the questions and answers generated by the large language model in *Going-deep* and correct the wrong answers for further refinement. The user may find that the LLM raises wrong questions, e.g., it asks whether the output list should be sorted. However, the desired output is an integer. In this case, the user must "Loop-back": review the specifications in *Paraphrase and Extend* to look for the wrong expressions leading to the wrong questions, then have them corrected. Loop-back is important because it is difficult to guarantee that the users never make mistakes.

3.4 Advantage Analysis

ChatCoder refines the original requirement mitigating vague and disambiguate expression, preventing the LLM from misunderstanding the human's requirement in downstream code generation. The dialogue scheme provides a scaffold for the LLM to refine the requirement. The two-round design makes the refined requirement more informative and with fewer mistakes. The design of ChatCoder makes its output refined requirement effective in assisting the LLM to generate code. Besides, in ChatCoder most of the text is generated by the LLM and humans need only to edit, saving much human labor for requirement refinement.

4 Experiments

4.1 Datasets

We select two datasets for our experiments:

Sanitized-MBPP A widely-used dataset(Austin et al., 2021) for evaluating an LLM's programming skill with 257 Python programming questions. Though the task descriptions are claimed to be manually checked for disambiguation by the authors, they are not clear enough for the LLM to understand and program correctly based on our manual check. Then, Sanitized-MBPP becomes a good playground to evaluate how ChatCoder can improve an LLM's programming skill by refining the requirements.

HumanEval A widely-used dataset(Chen et al., 2021) for evaluating an LLM's programming skill with 164 Python programming questions. The task descriptions are more longer and more complete than those of *Sanitized-MBPP*, leaving a challenge for ChatCoder to find the remaining key blind points for refinement.

4.2 Baseline Models and Generation Configurations

We use **gpt-3.5-turbo** and **gpt-4** as our baselines. Both gpt-3.5-turbo and gpt-4 are used as both the base LLM in ChatCoder and the model for comparison without using ChatCoder to refine requirements.

For *HumanEval*, we perform greedy generation, which means the generation is zero-shot, and the sampling is performed only once with a temperature of 0. For *Sanitized-MBPP*, we perform 3-shot generation. For each task, we sample 20 programs with top_p=0.2 when evaluating models for gpt-3.5-turbo. As for GPT-4, because there is a calling rate limit and the calling fee is high, it is difficult and expensive to sample 20 programs for a programming task. So we sample one program for a programming task with temperature 0 like HumanEval. The version of GPT-4 is gpt-4-0613. The version of gpt-3.5-turbo is gpt-3.5-turbo-0613. For a fair comparison, we rerun all the baselines with the same prompts and our generation configuration rather than copy the results from the original papers.

4.3 Metrics

We report the test pass rate(Chen et al., 2021). For HumanEval and Sanitized-MBPP on GPT-4, we report pass@1. We report pass@1, pass@5, and pass@10 for the other settings.

4.4 Research Questions

To evaluate our proposed ChatCoder, we raise and investigate the following research questions:

1) How does ChatCoder perform compared with existing code generation models?

2) Is ChatCoder an efficient method for LLM and human users to communicate for requirement refinement?

3) How much improvement is brought by human involvement in ChatCoder?

4.5 RQ1: Code Generation Performances

RQ1 evaluates whether ChatCoder enhances an LLM's code generation ability through its combined approach of LLM and human cooperation on requirement refinement. Specifically, we use ChatCoder on gpt-3.5-turbo to get the refined requirement expressions for Sanitized-MBPP and HumanEval. Then we compare gpt-3.5-turbo and gpt-4's code generation performances based on the refined requirements with the performances based on the original requirement expressions. We don't use gpt-4 to get the refined requirements due to the high cost and limited access to gpt-4.

	HumanEval	Sanitized-MBPP		
	pass@1	pass@1	pass@5	pass@10
gpt-3.5-turbo	70.12%	57.04%	59.13%	59.75%
gpt-4	81.10%	66.15%	-	-
ChatCoder(gpt-3.5-turbo)	79.87%	71.25%	75.18%	76.25%
ChatCoder(gpt-4)	90.24%	76.65%	-	-

Table 1: Code Generation Performances

According to Table 1, ChatCoder significantly improved the generated code's execution accuracy, especially for Sanitized-MBPP. The refined requirement specifications provided by ChatCoder helped gpt-3.5-turbo to increase its pass@1 from 57.04% to 71.25%. Compared horizontally, for both gpt-3.5-turbo and gpt-4, the performance improvements on Sanitized-MBPP are more prominent than those on HumanEval. Compared to HumanEval, Sanitized-MBPP has simpler task descriptions, leaving more blind points for ChatCoder to discover to improve the overall code generation performances.

	HumanEval	Sanitized-MBPP		
	pass@1	pass@1	pass@5	pass@10
gpt-3.5-turbo	70.12%	56.95%	59.48%	60.48%
Free Paraphrase	78.05%	64.61%	66.17%	66.68%
Free QA	71.95%	66.47%	70.91%	72.00%
ChatCoder	**79.87%**	**71.25%**	**75.18%**	**76.25%**

Table 2: Communication Efficiency Comparison

4.6 RQ2: Communication Efficiency Evaluation

RQ2 evaluates whether ChatCoder is sufficiently efficient for the LLM and the human to cooperate to refine requirements. ChatCoder should provide a dialogue scheme generating refined requirements which results in better downstream generated code than the other dialogue schemes for requirement refinement.

We compare ChatCoder with two other dialogue schemes for communication between human programmers and the LLMs: 1) **Free Paraphrase**: We let the large language model paraphrase the original programming task without giving any angles and ask the human user to have it edited and corrected for cognition alignment. 2) **Free QA**: We let the large language model ask human users questions about their confusion about the original programming task and collect the human users' responses. Comparing with these schemes helps evaluate whether the angles to extend requirements serve as scaffolds for the LLM to better understand the human requirements and the two-round dialogue setup, resulting in more concrete and informative requirment expressions. All these experiments are conducted based on gpt-3.5-turbo-0613. The results are presented on Table 2

According to Table 2, all three dialogue schemes help the LLM generate more accurate programs downstream, which implies that requirement refinement is effective in helping the LLMs in code generation. ChatCoder's improvement is more prominent than Free Paraphrase and Free QA due to its carefully designed dialogue scheme. The angles serve as scaffolds for the LLM to analyse and understand human requirements in requirement refinement and code generation. The two-round dialogue scheme provides not only more chances of finding blind points but also a method to find mistakes. Surprisingly, in Free Paraphrase and Free QA, the LLM refines the requirements in part of our proposed angles but can not cover all of them. Thus, providing explicit guidance for analysing requirements for the LLM is vital.

	HumanEval	Sanitized-MBPP		
	pass@1	pass@1	pass@5	pass@10
gpt-3.5-turbo	70.12%	56.95%	59.48%	60.48%
Auto-Refine	68.90%	52.82%	56.30%	57.12%
ChatCoder	**79.87%**	**71.25%**	**75.18%**	**76.25%**

Table 3: Human Intervention Evaluation

4.7 RQ3: Human Intervention Evaluation

RQ3 aims to evaluate the importance of human-in-the-loop in ChatCoder. The experiment intends to prove that effective refinement of requirements should involve both the software provider and the software supplier, in this case, the human user and the large language model. Specifically, we compare ChatCoder's performance with 'Auto-Refine', which involves the large language model paraphrasing and generating questions without human editing. All experiments are conducted using gpt-3.5-turbo-0613. The results are presented in Table 3.

The results show that Auto-Refine harms LLM's code generation performance even as a requirement refinement scheme. Because in Auto-Refine the LLM extends the requirements based on the knowledge from the training data different from the human user, the paraphrase can introduce errors and will not be corrected. Therefore, involving human users is necessary, and ChatCoder should be a human-in-the-loop technique.

5 Discussion

5.1 Case Study

We raise three cases illustrating how ChatCoder helps LLMs generate correct code rather than the wrong by refining requirements. The cases are shown in Figure 4.

MBPP/91 This task asks the coder to write a method checking if a string is presented in any string as a substring within a list. However, the large language model misunderstands the task of a list of words. ChatCoder corrects the misunderstanding. It refines the requirement expression by pointing out that the method should return a boolean value.

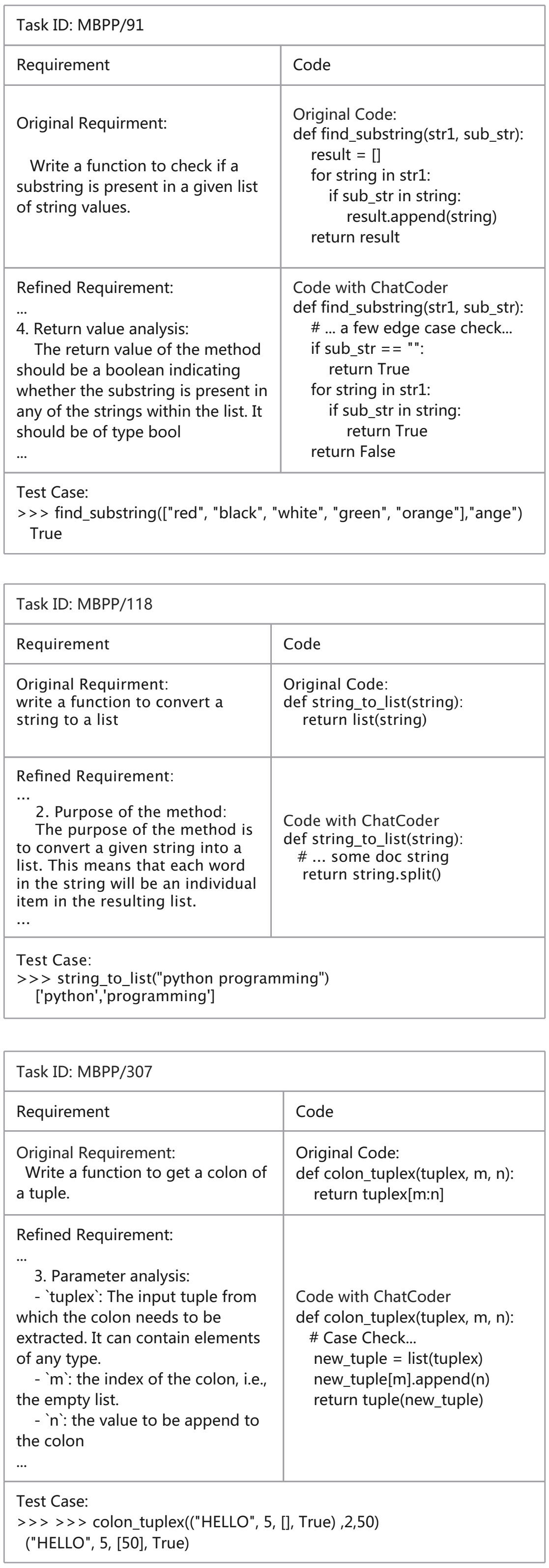
MBPP/118 This task asks the coder to write a method converting a string to a list. The LLM thinks the string is to be split into a list of characters, which should be split into a list of words. ChatCoder corrects the misunderstanding. It refines the requirement expression by explaining the concept 'split' to be 'split into words'.

MBPP/307 This task asks the coder to write a method to get a colon of a tuple. However, the meaning of the parameter is missing in the original requirement expression, causing the LLM to think that ‘m’ and ‘n’ are indexes. ChatCoder corrects the misunderstanding. It refines the requirement expression by directly explaining the meaning of the parameters.

With the refined requirements, the LLM generates the correct code in all three cases, showing the effectiveness of ChatCoder.

5.2 Savings of Human Labor Costs

Compared with performing requirements refinement with requirement engineers, ChatCoder asks the large language model to generate most of the text. At the same time, human users just need to review and edit, saving lots of human labour. This



Task ID: MBPP/91

Requirement | Code

Original Requirment:

Write a function to check if a substring is present in a given list of string values.

```
Original Code:
def find_substring(str1, sub_str):
    result = []
    for string in str1:
        if sub_str in string:
            result.append(string)
    return result
```

Refined Requirement:
...
4. Return value analysis:
The return value of the method should be a boolean indicating whether the substring is present in any of the strings within the list. It should be of type bool
...

```
Code with ChatCoder
def find_substring(str1, sub_str):
    # ... a few edge case check...
    if sub_str == "":
        return True
    for string in str1:
        if sub_str in string:
            return True
    return False
```

```
Test Case:
>>> find_substring(["red", "black", "white", "green", "orange"],"ange")
  True
```

Task ID: MBPP/118

Requirement | Code

Original Requirment:
write a function to convert a string to a list

```
Original Code:
def string_to_list(string):
    return list(string)
```

Refined Requirement:
...
2. Purpose of the method:
The purpose of the method is to convert a given string into a list. This means that each word in the string will be an individual item in the resulting list.
...

```
Code with ChatCoder
def string_to_list(string):
    # ... some doc string
    return string.split()
```

```
Test Case:
>>> string_to_list("python programming")
    ['python','programming']
```

Task ID: MBPP/307

Requirement | Code

Original Requirement:
Write a function to get a colon of a tuple.

```
Original Code:
def colon_tuplex(tuplex, m, n):
    return tuplex[m:n]
```

Refined Requirement:
...
3. Parameter analysis:
- `tuplex`: The input tuple from which the colon needs to be extracted. It can contain elements of any type.
- `m`: the index of the colon, i.e., the empty list.
- `n`: the value to be append to the colon
...

```
Code with ChatCoder
def colon_tuplex(tuplex, m, n):
  # Case Check...
    new_tuple = list(tuplex)
    new_tuple[m].append(n)
    return tuple(new_tuple)
```

```
Test Case:
>>> >>> colon_tuplex(("HELLO", 5, [], True) ,2,50)
  ("HELLO", 5, [50], True)
```

Figure 4: Case Study

section will analyse how much human labour costs are saved.

We evaluate the savings of the human labour costs by calculating how many tokens in the final refined requirement specifications are from humans. The statistics are shown in Figure 5. From Figure 5, we can see that tokens from human users take only a tiny proportion of the refined specifications. To boost the code generation performance, the users need to review the text, delete anything they do not like, and input, on average, ten tokens with the help of ChatCoder.

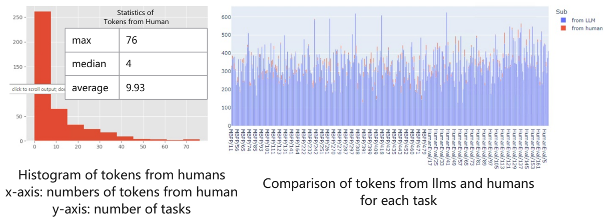


Figure 5: Statistics of Human Labor Savings

5.3 Relevance and Completeness

We need to evaluate whether the improvement is due to ChatCoder's refined requirements and whether the users think ChatCoder's refined requirement specifications fulfil their needs well. Thus, we invited three people outside the research group to give scores on ten randomly selected ChatCoder's refined requirements about 'relevance' and 'completeness'. The results are depicted in Figure 6. We ask the testers to compare the requirements before and after refinement and the code generated before and after the requirement refinement. Then, we ask them to give a score (1-5) to judge whether the refinement relates to the improvement of the generated code (The real score, 1 for unrelated and 5 for directly related). Besides, we ask them to give a score (1-5) to judge whether the refinement makes them clearer about the user's requirements (The comp score, 1 for getting confused and 5 for getting clear). We calculate the average scores with error bars and have the results depicted in Figure 6.

Through Figure 6, we find that all testers agree that the refined requirements help the large language model generate better code and help themselves better understand the requirements. However, compared with the real score, the confidence that people get clearer about the problems is slightly weaker. That is because people judge the quality of the code partially based on the execution test results. However, execution tests are not perfect. The program passing certain test cases may not fulfil the user's requirements. So, ChatCoder still needs to be improved to refine the requirements better to fulfil the user's actual needs.

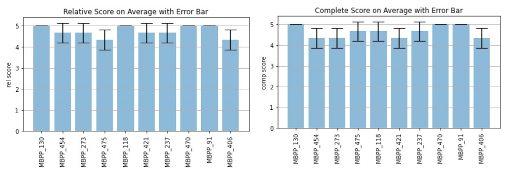


Figure 6: Human Evaluation Score

5.4 Threats to Validations

The choice of datasets to test ChatCoder is the major threat to our method. We use Sanitized-MBPP and HumanEval since they are popular and widely used code-generation datasets for evaluating LLMs. However, the real-world requirements and programs are more complex than the two datasets we use. To refine the more complex real-world requirements, it would be better to further improve the dialogue schemes in ChatCoder. Besides the choice of datasets, the experiment participants are another threat. The participants for this paper are all volunteer professional programmers who can give proper responses in the requirement refinement chat. However, the trait can not be guaranteed to be owned by any user of ChatCoder. Both threats can be evaluated by large-scale deploying ChatCoder in the real world and collecting user feedback which remains our future work.

6 Conclusion

We find that poorly expressed requirements degrade the LLMs' code generation performances. Then we propose ChatCoder to improve the LLM's code generation by introducing the human-in-the-loop requirement refinement via chat. With the carefully designed dialogue scheme, the LLM and the human user can disambiguate the requirements and help the LLM generate code more precisely. The experiments show that ChatCoder can not only improve the LLM's code generation ability by a large margin, better than other refinement techniques, and also save human labour.

References

1998. Ieee recommended practice for software requirements specifications. *IEEE Std 830-1998*, pages 1–

40.

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.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri 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, Josh 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.

E. de Jong, J. van de Pol, and J. Hooman. 2000. Refinement in requirements specification and analysis: a case study. In *Proceedings Seventh IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS 2000)*, pages 290–298.

Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Harkirat Singh Behl, Xin Wang, Sébastien Bubeck, Ronen Eldan, Adam Tauman Kalai, Yin Tat Lee, and Yuanzhi Li. 2023. Textbooks are all you need.

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

Jia Li, Yunfei Zhao, Yongmin Li, Ge Li, and Zhi Jin. 2023a. Towards enhancing in-context learning for code generation. *CoRR*, abs/2303.17780.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023b. Starcoder: may the source be with you!

Shaoying Liu. 2002. Capturing complete and accurate requirements by refinement. In *Eighth IEEE International Conference on Engineering of Complex Computer Systems, 2002. Proceedings.*, pages 57–67.

WenQian Liu. 2008. A requirements refinement framework. In *Proceedings of the 2008 ACM Symposium on Applied Computing (SAC), Fortaleza, Ceara, Brazil, March 16-20, 2008*, pages 658–659. ACM.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *CoRR*, abs/2306.08568.

OpenAI. 2023. GPT-4 technical report. *CoRR*, abs/2303.08774.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. *CoRR*, abs/2305.07922.