Chain of Targeted Verification Questions to Improve the Reliability of Code Generated by LLMs

Sylvain Kouemo Ngassom
Polytechnique Montréal
Montreal, Canada
sylvain.kouemo-ngassom@polymtl.ca

Arghavan Moradi Dakhel
Polytechnique Montréal
Montreal, Canada
arghavan.moradi-dakhel@polymtl.ca

Florian Tambon
Polytechnique Montréal
Montreal, Canada
florian-2.tambon@polymtl.ca

Foutse Khomh
Polytechnique Montréal
Montreal, Canada
foutse.khomh@polymtl.ca

ABSTRACT

LLM-based assistants, such as GitHub Copilot and ChatGPT, have the potential to generate code that fulfills a programming task described in a natural language description, referred to as a prompt. The widespread accessibility of these assistants enables users with diverse backgrounds to generate code and integrate it into software projects. However, studies show that code generated by LLMs is prone to bugs and may miss various corner cases in task specifications. Presenting such buggy code to users can impact their reliability and trust in LLM-based assistants. Moreover, significant efforts are required by the user to detect and repair any bug present in the code, especially if no test cases are available. In this study, we propose a self-refinement method aimed at improving the reliability of code generated by LLMs by minimizing the number of bugs before execution, without human intervention, and in the absence of test cases. Our approach is based on targeted Verification Questions (VQs) to identify potential bugs within the initial code. These VQs target various nodes within the Abstract Syntax Tree (AST) of the initial code, which have the potential to trigger specific types of bug patterns commonly found in LLM-generated code. Finally, our method attempts to repair these potential bugs by re-prompting the LLM with the targeted VQs and the initial code. Our evaluation, based on programming tasks in the CoderEval dataset, demonstrates that our proposed method outperforms state-of-the-art methods by decreasing the number of targeted errors in the code between 21% to 62% and improving the number of executable code instances to 13%.

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**; • **Software and its engineering** → **Software testing and debugging**.

AIware '24, July 15–16, 2024, Porto de Galinhas, Brazil

ACM ISBN 979-8-4007-0685-1/24/07
https://doi.org/10.1145/3664646.3664772

KEYWORDS

Large Language Model, Software Development, Reliability, Code Generation, Hallucination

ACM Reference Format:
Sylvain Kouemo Ngassom, Arghavan Moradi Dakhel, Florian Tambon, and Foutse Khomh. 2024. Chain of Targeted Verification Questions to Improve the Reliability of Code Generated by LLMs. In *Proceedings of the 1st ACM International Conference on AI-Powered Software (AIware '24), July 15–16, 2024, Porto de Galinhas, Brazil.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3664646.3664772

1 INTRODUCTION

The rapid advancements in Generative Artificial Intelligence (GenAI) offer different opportunities for its application across various domains, including Software Engineering (SE) [37]. Large Language Models (LLMs) like GPT4 [3], Codex [8], and Llama-2 [36] that can generate code based on a given natural language description, referred to as a prompt, have significantly advanced the automatic code generation process through their ability to synthesize code. The widespread accessibility of LLM-based assistant tools, such as ChatGPT [41], GitHub Copilot [17], and Copilot Chat [9] allows individuals with different backgrounds to generate code and contribute to open-source projects [23, 40]. However, there are still significant gaps between the code generated by LLM-based assistant tools and the code written by human software developers in terms of comprehension and quality [25, 34, 38].

The range of bugs in code generated by LLMs may vary from those easily detectable by an IDE linter to those that are obscure and challenging to identify [26, 35]. Presenting such buggy code to end users by LLM-based assistant tools may increase the efforts on the user's side to detect and repair these bugs, in a way that some users prefer to rewrite the code from scratch rather than incorporating suggestions from LLM-based assistant tools [38]. Also, users may, at times, not be aware of what a correctly generated solution should resemble, and the absence of test cases further complicates evaluating LLM-generated code and detecting their bugs [4, 25]. Thus, detecting and repairing those bugs in LLM-generated code before presenting them to the user can significantly improve their reliability.

Crafting effective prompts can be a solution to reduce human involvement and improve the reliability of LLM-generated code [19].

However, available solutions, such as incorporating the error messages raised by the compiler into the prompt [27, 30, 33] or using Automatic Programming Repair (APR) tools to repair the buggy code generated by LLM [11, 16], require test cases to trigger bugs in the code fragment. But, test cases may not always be readily available. Some studies suggest that iterative refinement can enhance the quality of code generated by LLMs [12, 20, 33]. However, these refinement steps are generally applied and may not effectively address specific bugs in the generated code, or they may necessitate the presence of test cases [20, 27].

In real software projects, developers typically pose verification questions to reduce ambiguity during the code review and improve the quality of submitted code [15, 18]. Additionally, raising questions about unexpected or incorrect aspects of students' solutions helps them to think around the incorrect part of their solutions and fill the gaps [32]. Inspired by these studies, we introduce a self-refinement method that leverages LLMs to improve the reliability of LLM-generated code by asking a chain of targeted Verification Questions (VQ), in the absence of test cases or concrete execution of the code.

Our method starts by converting the initial code generated by an LLM into an Abstract Syntax Tree (AST) [10]. Then, the method extracts features from targeted nodes in the AST that have the potential to trigger specific bug patterns. We choose the targeted nodes based on their properties which make them likely to lead to particular bug patterns as presented in existing LLMs' generated code bug taxonomy [35]. Subsequently, our method incorporates features of targeted nodes into different VQ templates generated by an LLM, depending on the node types. The approach then makes use of ChatGPT (gpt3.5-turbo) to iteratively improve the reliability of the initial code by prompting the model with the chain of VQs and repairing potential bugs. In the end, the method returns a repaired code where it fixed the targeted bug. The aim of our method, achieved through asking effective and targeted VQs, is to focus on a targeted bug while maintaining the correct code during the repairing process. In particular, our method does not require any prior execution of the code or a comprehensive set of test cases to repair the buggy code and improve the reliability of the initially generated code.

Our results indicate that our proposed method outperforms baseline methods, i.e., without any VQs or with general (non-targeted) VQs. In particular, our method reduces the amount of code generated with specific targeted error types up between 21% to 62% and improves the number of executable code instances generated by LLM by 13%. Furthermore, our approach introduces relatively few new bugs, around 12%, in the generated code (i.e. false positive) with the chain of VQs. Finally, while rephrasing the templates of targeted VQs can induce some variability in the output, the results remain consistent across rephrasing the template VQs differently in terms of their performance in bug repair.

To guide our study, we aim to answer the following RQs:

- **RQ1**: Do the chain of VQs repair the bugs in LLM-generated code?
- **RQ2**: Can VQs introduce defects in LLM-generated code?
- **RQ3:** How does rephrasing the template of VQs impact the performance of LLM in repairing the bugs?

The key contribution of this paper is:

- Proposing a self-refinement method that generates a chain of targeted VQs to first localize and then repair specific bugs in LLM-generated code before execution, without human intervention, and in the absence of test cases.

The targeted VQs in our study focus on the bug itself without replacing the entire code and it enables more control over the changes induced in the code during the repairing step. We are the first study that focuses on bug patterns that are prevalent in LLM-generated code. Our proposed method can be integrated as an agent that has been trained to enhance the reliability of LLM-generated code by generating and addressing a set of targeted VQs within an adversarial framework.

The rest of this paper is organized as follows: Section 2 presents the methodology. Section 3 describes the experimental setup along with the results. Section 4 discusses our results. Then, Section 5 details the threats to the validity of the study and Section 6 lists related works. Finally, the conclusion of the study is presented in Section 7.

2 METHODOLOGY

Our methodology is divided into four steps: ① First, the user leverages LLM to generate a code based on the prompt of the task, ② the method parse the obtained code AST, collecting features from relevant nodes, ③ based on the templates of VQs, targeted questions are generated by incorporating features from previously extracted nodes, ④ finally, combining the questions with a few-shot prompt, the method queries ChatGPT on the specific task to obtain a repaired code which is presented to the user. A schematic description of the method is presented in Figure 1. In the following, we describe the different steps in more detail.

2.1 Initial Prompt

The method starts by prompting the LLM to generate code for a specific programming task and collect its initial output. The prompt consists of a description of the programming task in natural language. Figure 1 ① illustrates an example of a function named "*match_pubdate*" generated by CodeGen in Python for a programming task described as "*Returns the first match in the pubdate_xpaths list*". This function aims to return the first node of an XML object that has a match in a given list of nodes. While our methodology is applicable across different programming languages, this study primarily focuses on the code generated by LLMs in Python as it is a very common and widely used programming language across different domains [1].

2.2 Extraction of AST Nodes

The Abstract Syntax Tree (AST) is an abstract representation of the parsing tree of code written in a programming language [10]. It is a tree data structure that captures the syntactic structure of the code while abstracting away details like punctuation and semicolons. Each node in the tree represents a syntactic construct of the code, such as expressions, statements, or declarations, and the tree's structure reflects the hierarchical relationships between these constructs. For example, Figure 1 ② shows a part of the AST dump of the function "*match_pubdate*" in Figure 1 ①. Each node in the

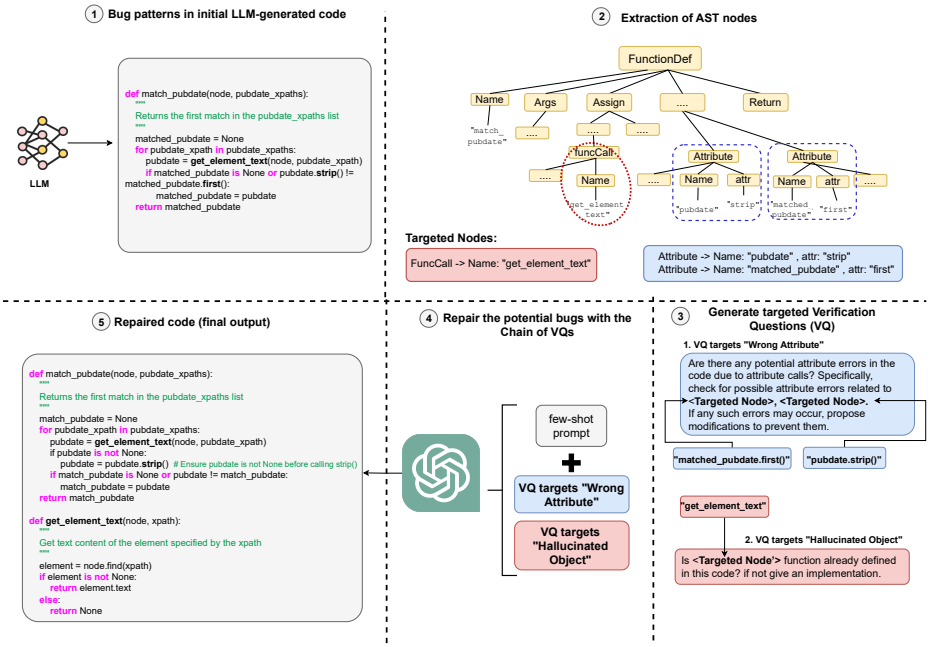

Figure 1: The proposed methodology for improving the reliability of code generated by LLMs in the absence of test cases.

AST also contains various features. For instance, the "*Attribute*" node in the AST shown in Figure 1 ② includes different features such as *name* and *attr*.

In the second step, the method localizes potential bugs by collecting different features from the initial LLM-generated code that may trigger specific types of errors. To localize the potential bugs, the method walks through the AST of the initial LLM-generated code and collects features on some targeted nodes that may trigger specific types of errors.

2.2.1 Bug Patterns in LLM-generated Code. Within the various types of bugs observed in LLM-generated code in Python, in this study, we focus on two specific categories identified in the LLM-generated codes' bugs taxonomy proposed by Tambon et al. [35]: "*Wrong Attribut*" and "*Hallucinated Object*". We narrowed down our investigation to these two types because, as indicated by the survey study on this taxonomy, participants highlighted them as patterns that are easy to detect but challenging to repair. However, our proposed method can be extended to address other bug patterns as well.

In the code example in Figure 1 ①, in the *if* condition, there are two attribute calls, "*strip()*" and "*first()*" (highlighted in bold). Such attribute calls hold the potential to trigger ***Attribute Error*** if they are invalid. In the absence of test cases, identifying whether these attribute calls can be accurately applied or if they might raise errors becomes challenging. For instance, in Figure 1 ①, "*strip()*" correctly operates as an attribute call on the variable "*pubdate*". However, "*first()*" is not a valid attribute in Python to be applied to the variable "*matched_pubdate*". According to the taxonomy of bug patterns in LLM-generated code [35], this type of bug is considered as "*Wrong Attribute*".

The code obtained in Figure 1 ① also incorporates a function called "*get_element_text*" (highlighted in bold). Such function calls hold the potential to trigger ***Name Error*** if they are not defined earlier. In this example, the model generates the code with the assumption that the "*get_element_text*" function should either be pre-defined or could be implemented. This kind of bug, while easy to notice, is hard to fix. In fact, this type of bug essentially requires implementing the primary objective of the task, which the model assumes is fulfilled by the hallucinated function. According to the taxonomy of bug patterns in LLM-generated code [35], this type of bug is classified as a "*Hallucinated Object*".

These two bug patterns can be linked to specific nodes in the AST. For example, the bug pattern "*Wrong Attribute*" which can be identified by an *Attribute Error* in Python, is linked to the node "*Attribute*" in AST and the bug patterns "*Hallucinated Object*" which can raise *Name Error* in Python, is linked to the node "*Call*" with *func* feature. The colored box in Figure 1 ② shows the features collected by our method on these targeted nodes.

2.3 Generating Targeted Verification Questions (VQs)

In this step, the method generates targeted Verification Questions (VQs) by considering features collected from each targeted node—nodes with the potential to trigger specific types of error. To do so, we first design templates for VQs in a way to verify the correctness of features of the targeted nodes or repair them if they are inaccurate.

To obtain an appropriate VQ template for each targeted node, we began by manually crafting VQs based on the bug patterns in LLM-generated code and the relevant nodes extracted from its AST. For instance, we craft a template designed to check the "*Wrong Attribute*"

pattern and verify attribute calls for potential *Attribute Error*s. The template VQ is "*Can you verify that the following attribute calls will not generate attribute error: <Targeted Nodes>. If any attribute error may occur, repair the code.*", where *<Targeted Nodes>* will be replaced by a list of relevant features extracted from the "*Attribute*" nodes in the AST of the generated code.

To mitigate the bias in our hand-crafted VQs, we then leveraged ChatGPT (gpt-3.5-turbo) [31] to generate final template VQs. To do so, we considered the hand-crafted VQs in a few-shot prompt and asked ChatGPT to generate similar template VQs. The few-shot prompt that we used in this step of the method can be found in our replication package [2]. This gives us general templates for the specific bug patterns we aim to repair. Figure 1 ③ shows the template VQs for two bug patterns in our study: "*Wrong Attribute*" and "*Hallucniated Object*", generated by ChatGPT. However, the targeted VQs can easily be expanded to other bug patterns in LLM-generated code by incorporating new VQ templates relevant to other bug patterns in LLM-generated code.

Our method used the template VQs to generate the targeted VQs by replacing the *<Targeted Nodes>* with the list of features collected from the AST of the LLM-generated Code, as shown in Figure 1 ③. For instance, in the VQ that targets "*Wrong Attribute*", the *<Targeted Nodes>* will be replaced by node features *matched_pubdate.first()* and *pubdate.strip()* to generate the targeted VQ. For the VQ that targets "*Hallucinated Object*", the *<Targeted Node>* will be replaced by *get_element_text()* which is an undefined function in this example and represents a hallucination generated by the LLM.

2.3.1 Rephrasing Templates of Targeted VQs. Studies indicate that rephrasing a prompt into semantically equivalent prompts can impact the output generated by LLMs [28, 29]. Even slight modifications, such as altering words with seemingly synonymous meanings, can lead to variations in LM outputs. To investigate the role of template VQs in our proposed method, we rephrase them by leveraging ChatGPT [31] for this rephrasing task. Drawing from previous research [29], we employed the ChatGPT and experimented with different instructions to prompt the model for the rephrasing task. For instance, Figure 2 represents the instruction that we used, inspired by Mizrahi et al. [29], to rephrase the template VQ relevant to "*Wrong Attribute*" bug pattern. Other instructions that we used in rephrasing the template VQs, along with the resulting rephrased template VQs are presented in our replication package [2].

2.4 Repair the Potential Bugs with the Chain of VQs on LLM

In the final step, the method aggregates all the targeted VQs generated in Section 2.3 into a chain of questions and attempts to use ChatGPT (gpt-3.5-turbo) to repair any bugs present in the initial LLM-generated code. We choose to use ChatGPT for repairing (so a different LLM than those that generated the initial code) as it allows for comparison across codes initially generated by different LLMs. Moreover, as the focus is on repairing an already generated code via targeted questions, leveraging a general model such as ChatGPT should yield better results than code-oriented LLMs.

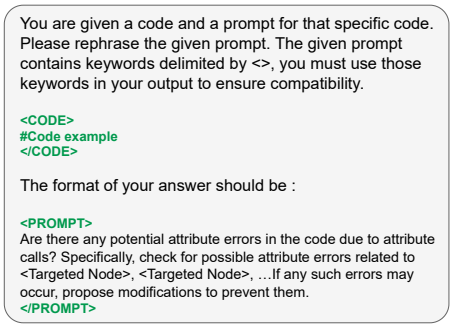

Figure 2: Instruction prompt to rephrase template VQ for "*Wrong Attribute*" bug pattern.

To prevent unnecessary alterations by the model when addressing the VQs or to avoid changing the correct targeted node, we employed a few-shot prompting strategy in this step (Figure 1 ④). The few-shot prompts include two samples of buggy code fragments (tagged by *<CODE>*), their relevant VQs (tagged by *<QUESTION>*), and the fixed code repaired only on the targeted nodes mentioned in the VQs (tagged by *<CORRECTION>*). These examples guide the model to focus on repairing any bugs present in the targeted nodes while leaving them unchanged if they are already correct. The last code fragment in the few-shot prompt represents the code under verification. While the examples in the few-shot prompt remain constant, the last code fragment and its relevant VQs will be updated with the new buggy code generated by the LLM and its corresponding VQs generated in step 3, Section 2.3. The few-shot prompt, along with its corresponding fixed examples used in our method, is already included in the replication package [2].

After invoking the prompt on ChatGPT, we expect the model to follow the VQs and provide, as a final output, a corrected version of the buggy code fragments. Figure 1 ⑤ illustrates a repaired version of the buggy code in Figure 1 ①, generated by ChatGPT using the chain of VQs prompt. It is noteworthy that while the final output may not be entirely correct, it has been already repaired on the targeted nodes that are buggy. For example, in Figure 1 ⑤, *pubdate.strip()* (considered in the VQ targeting the "*Wrong Attribute*") remains unchanged as it was correct while *matched_pubdate.first()* has been corrected since it was identified as buggy. Also, as shown in Figure 1 ⑤, the model has generated a function *get_element_text()* to repair the "*hallucinated object*" problem presented in Figure 1 ①.

3 EXPERIMENTS

3.1 Dataset and Environment Setup

We used the CoderEval [44] dataset in our experiments. The dataset is composed of 230 Python and 230 Java functions that were extracted from existing projects on GitHub. For each of those functions, the dataset contains the function's signature, the associated docstring, the function context (e.g., the complete file from which the function was extracted), and the actual extracted function which serves as an oracle. The dataset also contains 10 code samples, for

each function, generated by three different LLMs using the previously mentioned information. In our experiments, we focused on the Python tasks which had some of the generated samples labeled as "*Wrong Attribute*" or "*Hallucinated Object*" in the taxonomy by Tambon et al. [35] as those are the bug patterns we aim to correct. This gives us 36 individual tasks.

In CoderEval, to properly assess the validity of a generated code sample, a Docker environment was set which contains all complete project functions used in the dataset. To assess the validity of a generated code sample, the user just has to insert the generated code sample in the corresponding project within the environment and run the available tests. On average, a task in CoderEval has 4.5 tests with a minimum of 1 and a maximum of 27. Running the tests allowed for measuring whether the generated code was correct or not. For the **36** tasks we gathered, we recovered **61** buggy samples affected by the "*Wrong Attribute*" or "*Hallucinated Object*" bug patterns and **54** correct samples as flagged in the CoderEval replication package thanks to those tests. Those samples will be used as the initial code fragments in our experiments.

3.2 Experimental Setup

To answer each of our research questions, we design the following experiments:

3.2.1 RQ1: Do the chain of VQs repair the bugs in LLM-generated code? In this research question, we aim to assess whether the targeted VQs reduce the errors in the generated code. We will compare the results obtained with our method to two other methods: 1) a baseline without any verification questions, which simply lets the LLM generate the code without any refinement step (referred to as "No VQ" in the following), 2) another method has a refinement step based on asking a general verification question such as "*Can you improve this code or correct its bugs please ?*" inspired by existing approaches [16, 42] (referred to as "General VQ" in the following).

To compare the methods, we apply each of the methods to the 61 buggy samples of each task of CoderEval as we described in Section 3.1. We repeat the process five times each time with a different seed to obtain a fair comparison between methods and account for the stochasticity of the generation process. We then collected the generated code fragments and inserted them in the CoderEval environment as described in Section 3.1. Finally, we ran all the available tests. We collected whether or not each individual test ran on the generated code (whether or not there is an "AssertionError") or if an exception was raised (e.g., "AttributeError"). Thus, for each sample and each method, we have the results of each individual test which we split into: "Runnable" (the code runs whether or not there is an "AssertionError". This indicates that the code either successfully passed the test or it executed but failed to produce the expected output), "Attribute errors", "Name errors" and "Other errors" depending on the test outcome. It's worth noting that the "Other errors" category does not cover "Assertion errors", "Attribute errors" and "Name errors". As different tasks have different numbers of tests, we normalize the results per the number of tests. Furthermore, we aggregate those results at the sample level by considering the test results for a sample as follows: when the sample code runs for all tests (whether or not there are "AssertionErrors"), we consider the sample as "Runnable" and if at least one of the test yields an exception, we consider the sample of the relevant error type category. Note that, as we are automatically executing tests, samples manually flagged as "*Hallucinated Object*" and "*Wrong Attribute*" in the taxonomy, would only count towards one error type, as the test execution would raise and stop at one error (i.e. the earliest one) before executing the rest of the code sample and raising other errors in the code sample. In that case, we consider only the first raise error for the error type of that code sample in our experiments.

3.2.2 RQ2: Can VQs introduce defects in LLM-generated code? The goal of this question is to study the impact of our methods on correct code in more detail. That is, to see if the method does not introduce defects in the correct code. To answer this question, we took from the CoderEval dataset the 54 correct code samples generated by LLMs for the 36 individual tasks considered. We applied our described methodology to those code fragments and ran the tests on the obtained generated code after applying our methodology. Similarly to RQ1, we executed five runs by varying the seed and averaging the obtained results. In each case, we checked whether a correct code ended up being changed wrongly, that is, the method introduced a bug in the newly generated code.

3.2.3 RQ3: How does rephrasing the template of VQs impact the performance of LLM in repairing the bugs? The goal of this question is to study the impact of rephrasing the template of VQs on the output. We aim to assess if the repairing capability of the method is not hampered by how the templates of VQs are phrased. To do so, we generated five rephrasing versions of our initial template questions as explained in subsection 2.3.1 to see how rephrasing may affect the result. We used the same experiment process as RQ1 with our method by applying each of the rephrased prompts on the buggy samples.

3.3 Results

In the following, we present the results to answer each of our research questions.

3.3.1 RQ1: Do the chain of VQs repair the bugs in LLM-generated code? Results at the test-level are presented in Table 1 and results for the sample level are given in Table 2. "No VQ" and "General VQ" are the baseline methods and "Targeted VQs" presents the results of our proposed method.

Table 1: Average number of test cases resulting in a given error category. Results are normalized by the number of tests for a single task.

criteria	No VQ	General VQ	Targeted VQs
Runnable cases	0.2	22.42	**25.42**
Attribute errors	17.13	9.74	**5.8**
Name errors	18.13	6.24	**2.57**
Other errors	25.54	**22.6**	27.21

As observed in Table 1 and Table 2, our method, "Targeted VQs" performs better repair compared to the baseline, "No VQ" (before refinement): the number of tests not leading to an error type drastically diminishes for the targeted error types, "Attribute errors"

Table 2: Average number of samples resulting in a given error category

criteria	No VQ	General VQ	Targeted VQs
Runnable code	0	22.32	**25.33**
Attribute errors	16	9.64	**5.7**
Name errors	17	6.24	**2.39**
Other errors	28	**22.8**	27.58

and "Name errors", to 5.8 and 2.57 respectively. It also slightly diminishes the number of "Other errors". Moreover, the number of runnable tests increases from 0.2 for the "No VQ" method to 25.42 for the "Targeted VQs" method. We obtain similar trends at the sample level. Note that, at the test-level (Table 1), we obtain some tests that run while the sample code is buggy. This is because, for a few samples, the test case would trigger a branch that would not contain the bug. However, the sample itself remains buggy.

Our method, "Targeted VQ" shows improvement compared to the "General VQ" as well, mainly in terms of the targeted errors: "Attribute error" and "Name error". For example at the test level, the average number of tests with "Attribute errors" decreases from 9.74 to 5.8, and it decreases from 6.24 to 2.57 for the "Name errors". The number of "Other errors" increases, for example from 22.8 to 27.58 at the sample level. However, this cannot be controlled by the targeted VQs and so is possible that this is a side effect of the targeted VQs that may induce new bugs while repairing targeted bugs. We study this side effect in more detail in RQ2. For the "Runnable Code" category for example at the sample level (Table 2), the average number of runnable samples increased from 22.32 to 25.33. In conclusion, our proposed method, "Targeted VQ", translates by less code prone to "Attribute errors" and "Name errors" and slightly more "Runnable" code which was the goal of the chain of targeted VQs.

3.3.2 RQ2: Can VQs introduce defects in LLM-generated code? As shown in the results in Table 3, on average 6 correct code samples out of the 54 correct samples in our dataset turned to buggy ones by applying the chain of targeted VQs, which is a 12% false positive rate. Those errors are mainly either "other errors" (other than the targeted errors), 3.8 on average, or "Assertion errors" (the code samples execute correctly but fail on at least one of the tests), 2 on average. The number of code that ends up having an "Attribute" or "Name" error is very low, 0.2 on average. As such, the chain of targeted VQs with a few-shot prompt as explained in subsection 2.4, does not introduce many errors and, in particular, does not introduce the targeted errors. This demonstrates that while our method diminishes the number of targeted errors as indicated by the results in RQ1, in the absence of test cases where we are unable to discern the correctness of LLM-generated code, it will not erroneously turn too many correct code samples to the buggy ones (resulting in low false positives).

3.3.3 RQ3: How does rephrasing the template of VQs impact the performance of LLM in repairing the bugs? Table 4 shows the results of the impact of rephrasing at the sample level. We can observe that the different rephrasing introduces variability in the results. The sensibility of the LLMs to the prompt was already observed in previous studies [28, 29]. However, such rephrasing does not introduce a high variability in the results: for instance, the average number of runnable samples remains 25 while the average number of samples with "Attribute" or "Name" errors remains around 6 and 2.6 respectively. As such, while not immune to prompt rephrasing, our method is not highly impacted by the phrases in the prompt.

Table 3: Average number of correct codes which are wrongly changed by our method. Results are given at sample level.

Error Type	Average # of samples
From correct code to Assertion errors	2
From correct code to Attribute errors	0.2
From correct code to Name errors	0.2
From correct code to Other errors	3.8
Staying correct	47.8

4 DISCUSSION

Our study is the first study that focuses on bug patterns that are common in LLM-generated code, such as hallucination in function calls or attributes. It can localize potential bugs in LLM-generated code in the absence of test cases by compiling features from the AST of the generated code and then attempting to repair them using different instructions as a chain of targeted VQs. While bugs in LLM-generated code may result in the same error types (i.e. when using the Python interpreter) as those in human buggy code, the actions required to repair them differ from those applied to human buggy code. For example, a "*Hallucinated Object*" is a type of bug pattern that triggers a *Name Error* in Python. However, repairing this bug requires generating a new function that implements the primary objective of the task, similar to Figure 1 (1), which is not a typical bug pattern seen in human-written code. Our method is a self-refinement method that focuses primarily on enhancing the reliability of LLM-generated code by repairing its bugs before execution, without human intervention, and in the absence of test cases. This makes the approach particularly appealing compared to previous studies that are dependent on a set of comprehensive test cases or retrieval/fine-tuning-based methods [21, 24] which require a lot of data. On the contrary, our approach does not require any external data and relies on VQs which can be adapted for any kind of bug and the performance of LLMs in synthesizing code. Thus, our method can reduce the effort required for detecting and repairing bugs in LLM-generated code on the user's side in the absence of test cases and before execution. Moreover, it emphasizes trustworthiness towards the user, as it helps in preventing the said user from being confronted with buggy code [6].

Our proposed method, asking targeted VQs around potential bugs in code generated by LLM, is not limited to specific bug types studied in this paper and can be adapted to a diverse range of bug patterns. Our proposed method can be employed to generate a set of templates of targeted VQs for other bug patterns to enhance the reliability of LLM-generated code more comprehensively. Moreover, our method can be adapted for various improvements in LLM-generated code, including in quality, coding style, and comprehension. Our approach could particularly be helpful for those with less programming experience who may struggle to detect and repair bugs in code generated by the LLMs, sometimes overlooking

Table 4: Impact of rephrasing on the performances at a test normalized level evaluation

criteria	Runnable cases	Attribute errors	Name errors	Other errors
Rephrasing 0	25.42	5.8	2.57	27.21
Rephrasing 1	25.04	5.31	2.84	27.81
Rephrasing 2	24.85	6.66	2.42	27.07
Rephrasing 3	24.99	5.73	2.69	27.59
Rephrasing 4	25.09	6.47	2.83	26.31
Rephrasing 5	25.79	7.09	2.3	25.82

certain errors [11]. Finally, the targeted VQs in our study focus on the bug itself without replacing the entire code. In our experiments, we noticed that the general VQ, one of the baselines, led the model to drastically alter the code instead of repairing a specific bug, sometimes replacing the buggy code with an entirely new code, while our proposed method enables more control over the changes induced in the code.

However, our study only focused on the templates of VQs generated based on an initial predetermined structure and only investigated the effects of rephrasing on this template, as it was shown to have an impact on the LLMs output [29]. Further investigations are necessary to explore the implications of using different instructions within the template of VQs. Moreover. the templates are manually crafted before being adapted by ChatGPT. However, it would be possible to have an autonomous agent verifier comment to automatically generate and tailor the chain of VQs based on the initial generated code. Those considerations are left to future works.

5 THREATS TO VALIDITY

Internal Validity: The format of the templates for VQs in our study is predetermined to target specific types of bugs and the internal validity is mainly due to whether the templates of targeted VQs had the intended effect. We employ ChatGPT to generate the final targeted VQs by using the predetermined structured template alongside a few-shot prompt to avoid potential bias of the VQ being crafted by a human. However, the predetermined structure and format of the VQ templates by a human might impose limitations on the VQs generated by ChatGPT. To mitigate the effect of VQs, we compared our approach to two baselines (no VQ and general VQ) and repeated the experiment with each sample five times to account for non-determinism in the generation. Moreover, as recent studies highlighted that prompt rephrasing could impact LLMs' results [28, 29], we also studied the impact of rephrasing the templates of VQs on the results, but we didn't study the impact of different predetermined structured templates on the VQs generated by ChatGPT which can be discovered by future studies. The choice of the CoderEval dataset may also pose a threat to internal validity, given that it was chosen to leverage the taxonomy by Tambon et al. [35], which serves as the foundation for the bug patterns examined in our study. However, the advantage of this benchmark lies in its real programming tasks and methods extracted from GitHub repositories. This aspect highlights the usefulness of our method on more practical programming tasks compared to those tasks in more classical benchmarks such as MBPP [7] and HumanEval [8].
Construct Validity: As evaluation metrics, we consider the reduction of error types triggered by the targeted bug patterns in our study (*Attribute Error* and *Name Error*), along with the capacity of the code to be *Runnable* – that is, the code compiled without errors when provided with proper test inputs, even if the output isn't entirely correct. Furthermore, we compared the results of our method with two other baselines using these evaluation metrics. Additionally, we investigated whether our method introduces bugs into correct code (i.e. a form of "false positive") by examining the errors present in originally correct code after the application of our method. However, further investigation within a more comprehensive study is necessary to compare the regression among different types of errors in different code fragments by applying our method and other baselines.
External Validity: In our study, we mainly focus on code samples generated by three LLMs (PanGu-Coder, CodeGen, and Codex), as described in the CoderEval dataset. This could hamper the generalization of our study. However, we argue that our approach is model-independent, given its focus on the AST of the generated code. While the initial code fragments were generated by different LLMs in our study, our method relies only on ChatGPT for generating the final targeted VQs and repairing buggy code by addressing the chain of VQs. Our approach may benefit from ChatGPT's performance over other models used for the initial code generation, but future studies can explore alternative models to gauge their impact on our findings. In this study, our focus was on two bug patterns from Tambon et al. [35] taxonomy, namely "*Hallucinated Object*" and "*Wrong Attribute*". We chose these patterns because they rank among the most prevalent in the taxonomy. However, our proposed method can be extended to cover other bug types within its chain of VQs. Finally, we focused on code fragments generated in Python, given its widespread usage as one of the most prevalent programming languages, and these samples are used within the taxonomy. Nonetheless, we expect that similar targeted VQ could be effectively generated for other programming languages.
Reliability Validity: In our study, we focused on code fragments from CoderEval that were labeled either as "*Hallucinated Object*" or "*Wrong Attribute*" in Tambon et al. [35]. We verified that the samples led to such bug patterns before using them based on the dataset available on the taxonomy[35]. To enable other researchers to replicate or expand upon our study, we provide a replication package [2]. However, the ongoing enhancement of LLMs might pose challenges to achieving an exact replication of our findings.

6 RELATED WORKS

Some of the previous studies used APR tools to repair the buggy code generated by LLMs in order to enhance the reliability of their generated code [11, 16]. For instance, Fan et al. [16] employed an

APR tool to first detect and then repair the buggy code generated by Codex for specific LeetCode problems. Additionally, they re-prompt Codex with various single instruction prompts, such as "provide a fix for the buggy program" or "fix line N", to repair the bug in its initial generated code. Their findings indicate that leveraging Codex to repair its buggy-generated code performs better than an APR tool. This finding could be attributed to the fact that current APR tools are tailored to human bug patterns and may not be entirely effective on LLM-generated buggy code. Other studies, subsequent to generating the initial code by LLMs, re-prompt the model by incorporating additional information about the bugs into the prompt, such as error messages [13, 30, 33, 45]. Zhang et al. [46] investigated the potential of ChatGPT for program repair. Their results, based on a dataset of programming tasks, demonstrate that integrating error type and error line into the prompt enhances the repair capability of ChatGPT compared to a general instruction prompt like "repair the bug in the program". Another study proposed a fully autonomous code generation method by leveraging LLMs and adding the failed test cases into the prompt [27]. Their method first collects the test cases that failed on the code fragments generated by Codex. Then they construct a repair instruction by appending the input test to the prompt and re-prompting the LLM to repair the code in a manner that generates the expected test output based on the relevant test input. Their results outperform both the traditional genetic programming approaches and the initial output of Codex in generating correct code fragments. Another type of information that studies have shown to be useful for leveraging LLMs for program repairs is, adding relevant examples in the form of few-shot prompts [5, 21, 45]. For example, Joshin et al. [22] used the similarity between the error message of the target buggy program and the error message of a pool of buggy programs and their fixed versions, to collect relevant examples and added them into the prompt as a few-shot of buggy programs and their fixes. They employed a static analysis tool to localize the bug and compile the error messages. Their results demonstrate an improvement in repairing bugs in human code (not LLM-generated) compared to an APR tool. COCO [43] investigated the robustness of LLMs by adding information collected from the AST of the code into the initial prompts. Due to a robust model, adding features collected from the AST of its initial generated code to the same prompt—such as adding the imported library used in the initially generated code—shouldn't alter the model's output and should maintain consistency with the semantics of the initial code. However, their results indicate that incorporating such information into the same prompt can indeed alter the output of the LLM and is a good technique to evaluate their robustness. A study by Dhuliawala et al. [14] aimed to reduce hallucination in the natural language content generated by LLMs by fact-checking various aspects of the generated responses. They accomplished this by posing verification questions about different facts in the generated responses, such as "Where was Clinton born?" Their method then independently posed these verification questions to the LLM and used the model's responses to adjust the initial response accordingly. Their method demonstrates an improvement in hallucinations in the responses generated by LLMs for certain questions. Similarly, a study by Wu [39] also used LLMs to generate clarification questions for a programming task to improve the correctness of generated code. The method presented these clarification questions to the user and incorporated the user's responses to gain a better understanding of the programming task. In contrast, our method improves the reliability of LLM-generated code by repairing its bugs without human intervention and in the absence of test cases. While tools such as static analysis tools can detect a limited range of bug types, they may fail to identify the diverse range of error types in the absence of test cases, such as attribute errors in Python. Moreover, using both APR tools and error message information requires a comprehensive set of test cases to detect and repair the bug.

7 CONCLUSION

In this study, we aim to enhance the reliability of code generated by different LLMs in the absence of test cases and without human intervention. We achieve this by generating a chain of targeted VQs and re-prompting an LLM to repair potential bugs present in the initially generated code. To localize the potential bugs in the code generated by an LLM, our method traverses the AST of the initial code and gathers features from nodes that may trigger specific bug patterns. Then, it incorporates those features into the templates of targeted VQs generated by LLM and re-prompts the model to repair potential bugs by addressing the chain of VQs. Our focus in this study is primarily on two bug patterns based on the taxonomy of bugs in LLM code: "*Wrong Attribute*" and "*Hallucinated Object*". However, our proposed method is not limited to these patterns and can be extended to address other bug patterns. Our findings demonstrate that employing the chain of targeted VQs to re-prompt the LLM can reduce the number of *Attribute Error* and *Name Error* in the initial LLM-generated code fragments by up to 40% and 62%, respectively, compared to alternative baselines. Furthermore, applying the chain of targeted VQs to code fragments that were originally correct resulted in less than a 12% conversion to buggy code. Additionally, the rephrasing of targeted VQs did not significantly impact the results of our method. In future works, we plan to extend the chain of targeted VQs to discover additional bug patterns in the code generated by LLMs. We also intend to investigate the effects of employing various structures for template VQs within our method. Furthermore, our proposed method can be integrated into future studies where an agent has been trained to enhance the reliability of LLM-generated code by generating and addressing targeted VQs autonomously in an adversarial setup.

ACKNOWLEDGEMENT

This work is partially supported by the Canadian Institute for Advanced Research (CIFAR), and the Natural Sciences and Engineering Research Council of Canada (NSERC).

REFERENCES

[1] 2020. Top Programming Languages 2020. https://spectrum.ieee.org/top-programming-language-2020.
[2] 2024. The Replication Package. https://github.com/ExpertiseLLM/Chain-Of-Targeted-AST-Verification-Questions.
[3] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).
[4] Anisha Agarwal, Aaron Chan, Shubham Chandel, Jinu Jang, Shaun Miller, Roshanak Zilouchian Moghaddam, Yevhen Mohylevskyy, Neel Sundaresan, and Michele Tufano. 2024. Copilot Evaluation Harness: Evaluating LLM-Guided Software Programming. *arXiv preprint arXiv:2402.14261* (2024).

[5] Toufique Ahmed and Premkumar Devanbu. 2023. Better patching using LLM prompting, via Self-Consistency. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1742–1746.

[6] Nadia Alshahwan, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. 2024. Assured LLM-Based Software Engineering. *arXiv preprint arXiv:2402.04380* (2024).

[7] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).

[8] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[9] Copilot-chat 2023. Copilot Chat. https://docs.github.com/en/copilot/github-copilot-chat.

[10] Baojiang Cui, Jiansong Li, Tao Guo, Jianxin Wang, and Ding Ma. 2010. Code comparison system based on abstract syntax tree. In *2010 3rd IEEE International Conference on Broadband Network and Multimedia Technology (IC-BNMT)*. IEEE, 668–673.

[11] Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C Desmarais, and Zhen Ming Jack Jiang. 2023. Github copilot ai pair programmer: Asset or liability? *Journal of Systems and Software* 203 (2023), 111734.

[12] Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and Michel C Desmarais. 2023. Effective test generation using pre-trained large language models and mutation testing. *arXiv preprint arXiv:2308.16557* (2023).

[13] Pantazis Deligiannis, Akash Lal, Nikita Mehrotra, and Aseem Rastogi. 2023. Fixing rust compilation errors using llms. *arXiv preprint arXiv:2308.05177* (2023).

[14] Shehzaad Dhuliawala, Mojtaba Komeili, Jing Xu, Roberta Raileanu, Xian Li, Asli Celikyilmaz, and Jason Weston. 2023. Chain-of-verification reduces hallucination in large language models. *arXiv preprint arXiv:2309.11495* (2023).

[15] Felipe Ebert, Fernando Castor, Nicole Novielli, and Alexander Serebrenik. 2019. Confusion in code reviews: Reasons, impacts, and coping strategies. In *2019 IEEE 26th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 49–60.

[16] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated repair of programs from large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1469–1481.

[17] GitHub-Copilot 2022. GitHub Copilot. https://github.com/features/copilot.

[18] Xiaofeng Han, Amjed Tahir, Peng Liang, Steve Counsell, and Yajing Luo. 2021. Understanding code smell detection via code review: A study of the openstack community. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 323–334.

[19] Ahmed E Hassan, Dayi Lin, Gopi Krishnan Rajbahadur, Keheliya Gallaba, Filipe R Cogo, Boyuan Chen, Haoxiang Zhang, Kishanthan Thangarajah, Gustavo Ansaldi Oliva, Jiahuei Lin, et al. 2024. Rethinking Software Engineering in the Era of Foundation Models: A Curated Catalogue of Challenges in the Development of Trustworthy FMware. *arXiv preprint arXiv:2402.15943* (2024).

[20] Shuyang Jiang, Yuhao Wang, and Yu Wang. 2023. SelfEvolve: A Code Evolution Framework via Large Language Models. *arXiv preprint arXiv:2306.02907* (2023).

[21] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. Inferfix: End-to-end program repair with llms. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1646–1656.

[22] Harshit Joshi, José Cambronero Sanchez, Sumit Gulwani, Vu Le, Gust Verbruggen, and Ivan Radiček. 2023. Repair is nearly generation: Multilingual program repair with llms. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37. 5131–5140.

[23] Majeed Kazemitabaar, Xinying Hou, Austin Henley, Barbara Jane Ericson, David Weintrop, and Tovi Grossman. 2023. How novices use LLM-based code generators to solve CS1 coding tasks in a self-paced learning environment. In *Proceedings of the 23rd Koli Calling International Conference on Computing Education Research*. 1–12.

[24] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems* 33 (2020), 9459–9474.

[25] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems* 36 (2024).

[26] Yue Liu, Thanh Le-Cong, Ratnadira Widyasari, Chakkrit Tantithamthavorn, Li Li, Xuan-Bach D Le, and David Lo. 2023. Refining ChatGPT-generated code: Characterizing and mitigating code quality issues. *ACM Transactions on Software Engineering and Methodology* (2023).

[27] Vadim Liventsev, Anastasiia Grishina, Aki Härmä, and Leon Moonen. 2023. Fully Autonomous Programming with Large Language Models. *arXiv preprint arXiv:2304.10423* (2023).

[28] Antonio Mastropaolo, Luca Pascarella, Emanuela Guglielmi, Matteo Ciniselli, Simone Scalabrino, Rocco Oliveto, and Gabriele Bavota. 2023. On the robustness of code generation techniques: An empirical study on github copilot. *arXiv preprint arXiv:2302.00438* (2023).

[29] Moran Mizrahi, Guy Kaplan, Dan Malkin, Rotem Dror, Dafna Shahaf, and Gabriel Stanovsky. 2024. State of What Art? A Call for Multi-Prompt LLM Evaluation. arXiv:2401.00595 [cs.CL]

[30] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. Adaptive test generation using a large language model. *arXiv preprint arXiv:2302.06527* (2023).

[31] John Schulman, Barret Zoph, Christina Kim, Jacob Hilton, Jacob Menick, Jiayi Weng, Juan Felipe Ceron Uribe, Liam Fedus, Luke Metz, Michael Pokorny, et al. 2022. ChatGPT: Optimizing Language Models for Dialogue. *OpenAI blog (2022)* (2022).

[32] Meghan Shaughnessy, Rosalie DeFino, Erin Pfaff, and Merrie Blunk. 2021. I think I made a mistake: How do prospective teachers elicit the thinking of a student who has made a mistake? *Journal of Mathematics Teacher Education* 24 (2021), 335–359.

[33] Marta Skreta, Naruki Yoshikawa, Sebastian Arellano-Rubach, Zhi Ji, Lasse Bjørn Kristensen, Kourosh Darvish, Alán Aspuru-Guzik, Florian Shkurti, and Animesh Garg. 2023. Errors are Useful Prompts: Instruction Guided Task Programming with Verifier-Assisted Iterative Prompting. *arXiv preprint arXiv:2303.14100* (2023).

[34] Claudio Spiess, David Gros, Kunal Suresh Pai, Michael Pradel, Md Rafiqul Islam Rabin, Susmit Jha, Prem Devanbu, and Toufique Ahmed. 2024. Quality and Trust in LLM-generated Code. *arXiv preprint arXiv:2402.02047* (2024).

[35] Florian Tambon, Arghavan Moradi Dakhel, Amin Nikanjam, Foutse Khomh, Michel C Desmarais, and Giuliano Antoniol. 2024. Bugs in Large Language Models Generated Code. *arXiv preprint arXiv:2403.08937* (2024).

[36] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).

[37] Rosalia Tufano, Antonio Mastropaolo, Federica Pepe, Ozren Dabić, Massimiliano Di Penta, and Gabriele Bavota. 2024. Unveiling ChatGPT's Usage in Open Source Projects: A Mining-based Study. *arXiv preprint arXiv:2402.16480* (2024).

[38] Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. 2022. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Chi conference on human factors in computing systems extended abstracts*. 1–7.

[39] Jie JW Wu. [n. d.]. Large Language Models Should Ask Clarifying Questions to Increase Confidence in Generated Code. ([n. d.]).

[40] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. 2023. Autogen: Enabling next-gen llm applications via multi-agent conversation framework. *arXiv preprint arXiv:2308.08155* (2023).

[41] Tianyu Wu, Shizhu He, Jingping Liu, Siqi Sun, Kang Liu, Qing-Long Han, and Yang Tang. 2023. A brief overview of ChatGPT: The history, status quo and potential future development. *IEEE/CAA Journal of Automatica Sinica* 10, 5 (2023), 1122–1136.

[42] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1482–1494.

[43] Ming Yan, Junjie Chen, Jie M Zhang, Xuejie Cao, Chen Yang, and Mark Harman. 2023. Coco: Testing code generation systems via concretized instructions. *arXiv preprint arXiv:2308.13319* (2023).

[44] Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Tao Xie, and Qianxiang Wang. 2023. CoderEval: A Benchmark of Pragmatic Code Generation with Generative Pre-trained Models. *arXiv preprint arXiv:2302.00288v1* (2023).

[45] Jialu Zhang, José Cambronero, Sumit Gulwani, Vu Le, Ruzica Piskac, Gustavo Soares, and Gust Verbruggen. [n. d.]. Repairing bugs in python assignments using large language models (2022). *URL: https://arxiv. org/abs/2209.14876, doi* 10 ([n. d.]).

[46] Quanjun Zhang, Tongke Zhang, Juan Zhai, Chunrong Fang, Bowen Yu, Weisong Sun, and Zhenyu Chen. 2023. A critical review of large language model on software engineering: An example from chatgpt and automated program repair. *arXiv preprint arXiv:2310.08879* (2023).

Received 2024-04-05; accepted 2024-05-04