An Empirical Study of the Non-determinism of ChatGPT in Code Generation

SHUYIN OUYANG, King's College London, United Kingdom
JIE M. ZHANG, King's College London, United Kingdom
MARK HARMAN, University College London, United Kingdom
MENG WANG, University of Bristol, United Kingdom

There has been a recent explosion of research on Large Language Models (LLMs) for software engineering tasks, in particular code generation. However, results from LLMs can be highly unstable; nondeterministically returning very different code for the same prompt. Such non-determinism affects the correctness and consistency of the generated code, undermines developers' trust in LLMs, and yields low reproducibility in LLM-based papers. Nevertheless, there is no work investigating how serious this non-determinism threat is.

To fill this gap, this paper conducts an empirical study on the non-determinism of ChatGPT in code generation. We chose to study ChatGPT because it is already highly prevalent in the code generation research literature. We report results from a study of 829 code generation problems across three code generation benchmarks (i.e., CodeContests, APPS, and HumanEval) with three aspects of code similarities: semantic similarity, syntactic similarity, and structural similarity. Our results reveal that ChatGPT exhibits a high degree of non-determinism under the default setting: the ratio of coding tasks with zero equal test output across different requests is 75.76%, 51.00%, and 47.56% for three different code generation datasets (i.e., CodeContests, APPS, and HumanEval), respectively. In addition, we find that setting the *temperature* to 0 does not guarantee determinism in code generation, although it indeed brings less non-determinism than the default configuration (*temperature*=1). In order to put LLM-based research on firmer scientific foundations, researchers need to take into account non-determinism in drawing their conclusions.

1 INTRODUCTION

Large Language Models (LLMs) are nondeterministic by nature [34]. This is because LLMs predict the probability of a word or token given the context, represented by a sample of words. The randomness in LLMs typically comes from the sampling methods used during text generation, such as top-k sampling or nucleus sampling [31, 50]. As a result, identical instructions or prompts can yield completely different responses to separate requests.

This non-determinism (i.e., the inconsistency in the code candidates generated in different requests with identical prompts)[1] is an essential consideration when using LLM in practice [59]. Unreliable and inconsistent code snippets can have significant negative effects on the process of software development, particularly in safety-critical applications where consistency and reliability are paramount [11, 30]. It may also undermine developers' trust in LLMs when completely different suggestions are given at different times [64].

[1]There are other terms in the literature that also refer to non-determinism, such as inconsistency, variance, randomness, and instability.

Authors' addresses: Shuyin Ouyang, King's College London, London, United Kingdom; Jie M. Zhang, King's College London, London, United Kingdom; Mark Harman, University College London, London, United Kingdom; Meng Wang, University of Bristol, Bristol, United Kingdom.

ACM 1557-7392/2024/9-ART
https://doi.org/10.1145/3697010

Moreover, non-determinism affects the reliability and reproducibility of empirical software engineering [54]. Indeed, compared to other tasks of ChatGPT, such as question answering and text summarization, the non-determinism threat in code-related tasks is much more serious, because the inconsistency (especially semantic inconsistency) often indicates errors in the generated code [28]. It is therefore of vital importance to understand how serious the non-determinism is for LLM-based software engineering tasks and call for actionable solutions to alleviate this issue.

This paper presents the first systematic empirical study on the threat of non-determinism of ChatGPT in code generation tasks. We choose the code generation tasks because code generation with Large Language Models (LLMs), such as ChatGPT, has recently attracted significant attention due to its impressive and cutting-edge performance [10, 15, 37]. Indeed, many publications have emerged from both the software engineering community and the machine learning community on evaluating the capability of ChatGPT in code generation [6, 10, 16, 41, 69].

This paper focuses on ChatGPT (including GPT-3.5 and GPT-4), rather than other LLMs, for the following two reasons: 1) ChatGPT is the most widely adopted LLM in code generation in the literature [15, 16, 23, 42, 44, 65, 72]; 2) ChatGPT has the best performance in code generation and represents the state-of-the-art so far [4, 15]. Thus, as the first work on the non-determinism of LLMs in software engineering tasks, we focus on ChatGPT in this paper but encourage other work to continue to investigate the non-determinism issue in other LLMs.

We conduct a series of experiments using the ChatGPT models on three widely-studied code generation benchmarks (i.e. CodeContests, APPS, and HumanEval) with 829 coding problems. For each code generation task, we let ChatGPT make five predictions. We then compare the similarity of the five code candidates from three aspects, namely *semantic similarity*, *syntactic similarity*, and *structural similarity*. We also explore the influence of temperature (i.e., a parameter that controls the randomness of the response generated by ChatGPT) on non-determinism, as well as the correlation between non-determinism and coding task features such as the length of coding instruction and the difficulty of the task. We show the non-determinism with different models of ChatGPT, namely, GPT-3.5 and GPT-4. Finally, we compare the non-determinism of code generation with different prompt engineering strategies.

Our results reveal that the threat of non-determinism in ChatGPT for code generation is serious, especially under default setting: In particular, 1) the ratio of problems with not a single equal test output among the top-five code candidates is above 50% for all the benchmarks we study; 2) the maximum difference of the test pass rate reaches 1.00 for all three datasets, and accounts for 39.63% of the problems in HumanEval, the most widely used code generation benchmark; In addition, contrary to the widely held belief (and practice followed to minimize nondeterminism) [7, 13, 39], setting the temperature to zero does not guarantee determinism in code generation. Also interestingly, our result analysis suggests that the length of coding instructions has a negative correlation with almost all our similarity measurements, meaning that longer description length tends to yield code candidates with less similarity and more buggy code. Different prompt engineering strategies also yield different degrees of non-determinism in code generation.

To understand how the literature handles the non-determinism threat, we collect 76 LLM-based code generation papers that appeared in the last 2 years. Our manual analysis results highlight that only 21.1% of these papers consider the non-determinism threat in their experiments. These results highlight that there is currently a significant threat to the validity of scientific conclusions. We call for researchers to take into account the non-determinism threat in drawing their conclusions.

To summarize, this paper makes the following contributions:

- We present the first study of the non-determinism threat in code generation tasks on ChatGPT, with three widely-studied datasets (CodeContest, APPS, HumanEval) and three types of similarity measurements. Our

results reveal that the non-determinism threat is serious and deserves attention from both academia and industry.
- We study the influence of temperature on the non-determinism of ChatGPT and find that setting temperature to zero does not guarantee determinism in code generation, which is contrary to many people's beliefs.
- We study the correlation between coding task features and the degree of non-determinism. The results reveal that the length of coding instruction has a negative correlation with syntactic and structural similarity, as well as the average correctness of the generated code.
- We study the influence of different prompt engineering techniques on code generation non-determinism. We find that prompts with a Chain-of-Thought strategy leads to more non-determinism when temperature=0, while code candidates generated from prompts requesting simple and concise code are more stable.

We release our data, code, and results at our homepage [3]. The rest of the paper is organized as follows. Section 2 introduces the main procedure of our study. Section 3 describes the design of the experiments, including research questions, benchmarks, selected models, and measurement tools. Section 4 presents the results and discusses some interesting findings based on the experimental results we obtained. Section 5 discusses the threats to validity in two aspects, as well as the limitations of this study. Section 6 introduces the related work of our study. Section 7 discusses the implications for software developers and researchers and future work. Section 8 concludes.

2 METHOD

Fig 1 shows an overview of our experimental procedure. For each code generation task, our study first produces a prompt with a coding instruction, then feeds this prompt to ChatGPT API [2] to generate code (zero-shot). We call the API five times to let ChatGPT make five predictions with the same prompt. We then extract code from each of the five responses, to get five code candidates. Our non-determinism analysis compares the five code candidates in terms of their semantic similarity, syntactic similarity, and structural similarity.

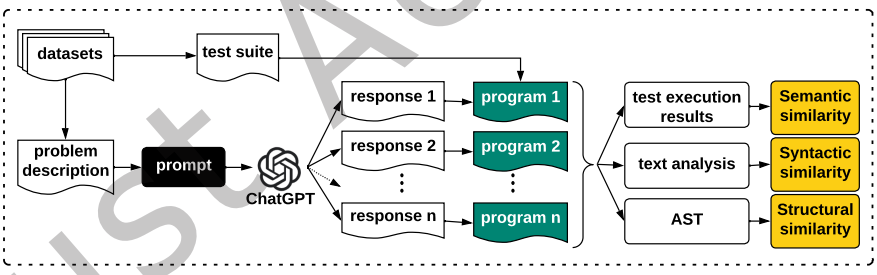


Fig. 1. Overview of the experimental procedure.

Prompt synthesis: The first step in our study is prompt preparation. There are many ways to conduct prompt engineering for code generation. In this paper, we follow the common practice in LLM-based code generation assessment [5, 15]. In particular, 1) we ask ChatGPT to generate Python code for each code generation task with zero-shot prompting; 2) we use the basic prompt design directly followed by programming task descriptions. To guarantee that ChatGPT produces code rather than pure natural languages in its response, we augment the original coding problem description with an instruction to request for Python code.

One challenge in extracting the code from the API response is that there is no clear signal to distinguish code with plain text in the response, which is different from ChatGPT's web chat window (i.e. in the chat window,

codes are returned with Markdown code blocks). To address this problem, we specify the format of the generated code into 'Markdown'. Thus, for each code generation task, our prompt is shown as follows:

```
Generate Python3 code (Markdown):
# this is the original coding problem description.
```

Code Extraction: After receiving the response from ChatGPT, we apply code extraction to retrieve the code from the generated text. We compile the code directly without making any modifications. Our experiments are mainly run on Google Deep Learning VM instances, with the Linux environment pre-installed from open images[2]. All of the necessary libraries are pre-installed. In this way, it can ensure to the greatest extent that the generated code will not cause import errors caused by the library not being installed during running.
Test Case Execution: To evaluate the semantics of ChatGPT's generated code, we use the test suite that is suited to each benchmark. We not only record whether each test passes or not but also record every specific test output, which enables us to compare the similarity of test outputs even if they both fail. For CodeContests and HumanEval datasets, every problem has a certain timeout value of 3 seconds. The APPS dataset does not provide a default timeout value, and we set the value to be 3 seconds as well. We use single-threaded scripts to run the tests to ensure that the test cases are executed sequentially to avoid race conditions that may arise from concurrent executions.
Similarity Checking: To measure the similarity between code candidates, we introduce similarity measurement tools that evaluated the semantic, syntactic, and structural similarity between the generated code solutions. The semantic similarity is measured by comparing test execution outputs. The syntactic similarity is measured by comparing the text similarity between codes. The structural similarity is evaluated by comparing the code candidates' abstract syntax trees (ASTs). More details about our similarity measurement methods are mentioned in Section 3.4.

3 EXPERIMENTAL DESIGN

3.1 Research Questions

This study answers the following questions:
RQ1: To what extent is ChatGPT susceptible to non-determinism in code generation under the default setting? This RQ investigates the non-determinism of ChatGPT in terms of the semantic, syntactic, and structural similarity among the code candidates generated with identical instructions under the default setting. There are three sub-RQs:

- *Sub-RQ1.1: To what extent is ChatGPT susceptible to non-determinism in terms of semantic similarity?*
- *Sub-RQ1.2: To what extent is ChatGPT susceptible to non-determinism in terms of syntactic similarity?*
- *Sub-RQ1.3: To what extent is ChatGPT susceptible to non-determinism in terms of structural similarity?*

RQ2: How does temperature affect the degree of non-determinism? Temperature is a hyperparameter of LLMs for controlling the randomness of the predictions. This RQ checks and compares the non-determinism of ChatGPT in code generation with different choices of temperature.
RQ3: How does the non-determinism compare to the similarity of the top code candidates generated within the same prediction? ChatGPT can be configured to generate multiple candidates for one prediction, which are ranked by their predictive probability. This RQ compares the similarity of the code candidates obtained in different predictions with those obtained within the same prediction.
RQ4: What types of coding tasks have a higher degree of non-determinism? To understand what affects non-determinism, this RQ studies the correlation between the features of coding tasks (e.g., the length of code

[2]https://cloud.google.com/compute/docs/images

generation instructions, the code problem difficulty, and labels) and the similarity metrics used in our study. We also conduct qualitative analysis on specific cases for deep analysis.

RQ5: How is GPT-4's non-determinism compared with GPT-3.5? This RQ compares GPT-3.5 and GPT-4 in their degree of non-determinism in generating code.

RQ6: How do different prompt engineering strategies influence the degree of non-determinism? This RQ compares the degree of non-determinism for different prompt engineering strategies (i.e., Chain-Of-Thought and requesting generated code as concise as possible) when using ChatGPT to generate code.

3.2 Code Generation Benchmarks

Our experiments use the three most widely studied code generation benchmarks: CodeContest [37], APPS [26], and HumanEval [12]. Table 1 shows their details. Each of these datasets has unique characteristics, which are introduced below. The distribution of difficulty and problem tags of these datasets are available on our homepage [3].

Table 1. Code generation benchmarks

Name	Mean Length of description	No. of Problems	Mean No. of Test Cases	Mean No. of Provided Correct Solutions
CodeContests	1989.19	165	203.84	49.99
APPS	1663.94	500	80.43	20.92
HumanEval	450.60	164	9.24	1.00

CodeContests: CodeContests is used when training AlphaCode, which comprises coding problems from various sources such as Aizu[3], AtCoder[4], CodeChef[5], CodeforcesCodeChef[6], and HackerEarthCodeChef[7]. In our experiment, following the assessment practice of AlphaCode, we use the test set of CodeContests to benchmark the code generation tasks of ChatGPT.

APPS: APPS includes 10,000 coding problems (both the training set and testing set). This dataset is exclusively designed for Python program synthesis evaluation. The original test set contains 5,000 code-generation problems, and we randomly sample 500 problems, among which there are 60.20% interview problems, 19.60% introductory problems, and 20.20% competition problems. APPS evaluates models not only on their ability to code syntactically correct programs but also on their ability to understand task descriptions and devise algorithms to solve these tasks [27].

HumanEval: The HumanEval dataset is an evaluation set first proposed in [12], which contains 164 hand-written coding problems. Each problem includes a function signature, docstring, body, and several unit tests, with an average of 9.24 test cases per problem. We use the whole dataset to benchmark our experiments.

As mentioned in Section 2, we especially focus on the code generated with Python3 language, since it is one of the most widely studied programming languages in code generation [5, 12, 17, 37, 61, 63, 66].

3.3 Configuration of ChatGPT

ChatGPT has gained widespread popularity and recognition in multiple tasks including question-answering, language translation, sentiment analysis, and text summarising, among which code generation is one of the most

[3]https://judge.u-aizu.ac.jp
[4]https://atcoder.jp
[5]https://www.codechef.com
[6]https://codeforces.com
[7]https://www.hackerearth.com

impressive tasks [10, 37]. There are several reasons why we have chosen ChatGPT as our research target among all large language models. Firstly, ChatGPT has the ability to generate highly coherent and contextually appropriate responses to a wide variety of textual prompts [25]. This makes it an ideal tool for conducting research in areas of code generation by designing specific prompts. Secondly, the GPT-3.5 series is a particularly attractive option due to its impressive performance and large-scale training data, which allows for more accurate and nuanced language processing capabilities [43]. Thirdly, the model API 'gpt-3.5-turbo' and 'gpt-4' released with ChatGPT have not been extensively studied in academia, and their capabilities in terms of code generation are thus still unknown. Therefore, we choose them as our experiment target models. Written in ChatGPT's official website[8], using ChatGPT's model API requires various parameters. We use the default values for most of the parameters in addition to the following ones:

- ***model***: ID of the model to use. This parameter is strictly required, and in our case, we set this parameter to 'gpt-3.5-turbo-0125' or 'gpt-4-0613'.
- ***message***: A list of messages describing the conversation so far, where two key values 'role' and 'content' should be filled. This parameter is also strictly required. In our experiments, the message's 'role' is 'user' and the 'content' contains the prompt we used for requesting for all of the RQs.
- ***temperature***: What sampling temperature to use, between 0 and 2 (Default value is 1). Higher values will make the output more random, while lower values will make it more focused and deterministic. In our study, we study the influence of temperature in RQ2 with three temperature values: 0, 1, and 1.5. For RQ1, we use *temperature*=1 only, and for the rest of the RQs, we present results with both *temperature*=1 and *temperature*=0.
- ***top_p***: An alternative to sampling with temperature, called nucleus sampling, where the model considers the results of the tokens with *top_p* probability mass. In our experiment, we do not take it into consideration and set this value to remain at its default setting (i.e., *top_p*=1).
- ***n***: How many code candidates (the so-called "chat completion choices" according to the ChatGPT API website [2]) to generate for each input message (with 1 being the default value). The default value of *n* is 1. In RQ3, we set *n*=5 to investigate how the non-determinism of code candidates from the same request compares with those from different requests. We choose *n*=5, since 5 is a widely used figure in the papers studying variance [51]. *n*=5 is only used in RQ3.

3.4 Non-determinism Measurement

In order to answer our research questions, we introduce the following tools for measuring the degree of non-determinism.

3.4.1 Semantic similarity. We measure the semantic similarity of different code candidates by checking their similarity in test execution results, including ***test pass rate*** and ***output equivalence rate***. The test pass rate calculates the ratio of the passed test case number against the total test case number for code candidates. It is one of the most widely used measurement metrics for assessing code generation capabilities[9] [5, 12, 26, 37, 73]. Each code generation problem has five test pass rates, one for each code candidate. We use the variance and maximum difference of the five values to indicate semantic similarity. We also calculate the mean of the five values for the purpose of understanding correctness as well as the correlation between correctness and non-determinism (RQ4).

The output equivalence rate records the ratio of identical test outputs (across different code candidates for the same code generation instruction) against the total test outputs. Each instruction has one output equivalence rate. For tests that produce specific outputs (without exceptions or errors), we check whether the output values of different code candidates are equal to each other. In the following parts of this paper, we use OER to represent

[8]https://platform.openai.com/docs/api-reference/chat/create

[9]Although the benchmarks are very widely studied, their test suites can be inadequate. This paper is less affected by the inadequate test suite issue as we focus on the similarity of test pass rate, rather than the absolute value of test pass rate.

output equivalence rate and use OER (no ex.) to represent output equivalence rate (without exceptions or errors) for short. Each code generation problem has only one OER and OER (no ex.). Additionally, we measure the OER and OER (no ex.) in pairs and report the mean output equivalence rate of the combinations of every two code candidates for a coding problem. For tests that yield exceptions or timeout errors, we consider the test outputs to be the same if the exception or error messages are the same.

Some papers use the pass@k metric [12, 32] (i.e., the ratio of coding tasks with 100% test pass rate) to indicate the high-level code generation correctness of a code generation approach. We do not use this metric in our main body of experiments because we focus on the non-determinism threat, while pass@k ignores the correctness of each single coding task and concentrates only on the ratio of correct code candidates in all the tasks, which can cover the non-determinism across different requests. In addition, pass@k does not reflect the practical application scenario of LLMs in code generation, because developers are less likely to try the model for k times until they finally get one correct solution.

3.4.2 Syntactic similarity. The syntactic similarity in this study treats different code candidates as texts and checks their textual similarity. We choose the **Longest Common Subsequence** and **Levenshtein Edit Distance** as evaluation tools [35, 36, 47, 68]. In the following content, we use LCS and LED to represent the Longest Common Subsequence and Levenshtein Edit Distance for short respectively. LCS measures the similarity via the normalized length of the longest common subsequence between two sequences. LED measures the minimum number of single-token edits (insertions, deletions, or substitutions) required to change one code into the other. LCS and LED both regard the token as the smallest unit, and the token is divided by the `.split()` method, that is, any whitespace is used as the separator to divide the code into tokens. We measure the syntactic similarity with LCS/LED by comparing the first code candidate with each of the remaining four code candidates. Thus, each code-generation problem has four values of each metric. We use the mean, mean worst value (i.e., mean highest value for LED and mean lowest value for LCS), and pair mean (by comparing all the combinations of two code candidates in pairs) to indicate the syntactic similarity measured by each metric.

Below are the formulas for the LCS and LED:

$$LCS = \frac{len(lcs(s,t))}{len(s)}$$

where s is reference string, t is the string to be compared, $lcs(s,t)$ is the longest common subsequence between s and t.

$$\text{LED}_{s,t}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) = 0 \\ \min \begin{cases} \text{led}_{s,t}(i-1,j)+1 \\ \text{led}_{s,t}(i,j-1)+1 \\ \text{led}_{s,t}(i-1,j-1)+1_{(s_i \neq t_j)} \end{cases} & \text{otherwise} \end{cases}$$

where $\text{LED}_{s,t}(i,j)$ is the LED between the first i characters of s and the first j characters of t, and $\text{diff}(s_i, t_j)$ is 0 if the i-th character of s is the same as the j-th character of t, and 1 otherwise.

3.4.3 Structural similarity. We design structural similarity to measure the code similarity in terms of the Abstract Syntax Tree (AST). AST is a tree-like representation of the source code in which each node in the tree represents a construct in the code, such as variable, function, or control structure, and the edges between nodes represent the relationships between these constructs. We use a Python library called `pycode_similar`[10] [38, 67] to calculate the similarity. The `pycode_similar` normalizes Python code into AST representation and uses Python library `difflib` to get the modification from referenced code to target code. There are two different measurement settings, i.e. **Unified_Diff** and **Tree_Diff**. Unified_Diff measures the difference of normalized function AST

[10]https://github.com/fyrestone/pycode_similar

string lines, while Tree_Diff measures the difference in tree edit distance between two given ASTs. Similar to syntactic similarity, for each code generation problem, we report the mean, smallest similarity values, and pair mean among the five candidates.

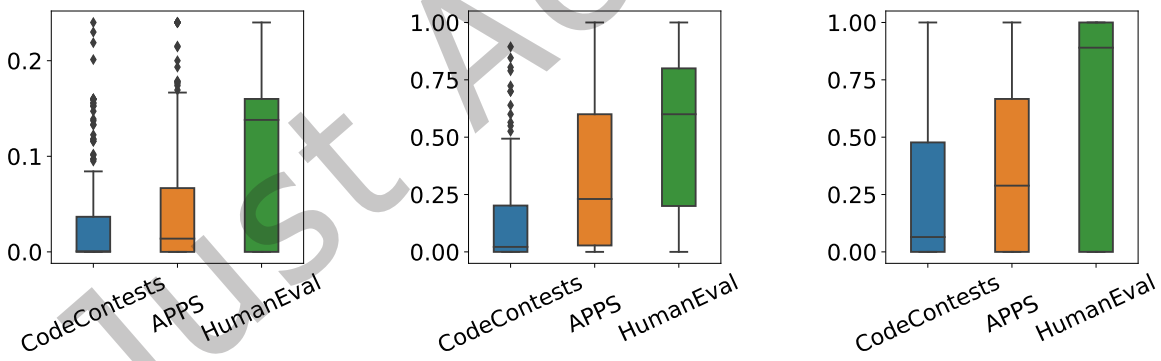
3.4.4 Statistical Analysis. We conduct statistical analysis to demonstrate the significance of the differences among the outputs. We choose Kruskal-Wallis test [48] which does not require assumptions of normal distribution. The Kruskal-Wallis test stands as a non-parametric method for analyzing data, serving as an extension of the Mann-Whitney U test [49] to more than two independent groups. The essence of the Kruskal-Wallis test lies in comparing the median ranks among groups, rather than the means, which makes it robust against outliers and non-normal distribution of data.

4 RESULTS AND FINDINGS

This section introduces the experimental results as well as the analysis and discussion for each RQ.

4.1 RQ1: Non-determinism of ChatGPT with Three Types of Similarities under default setting

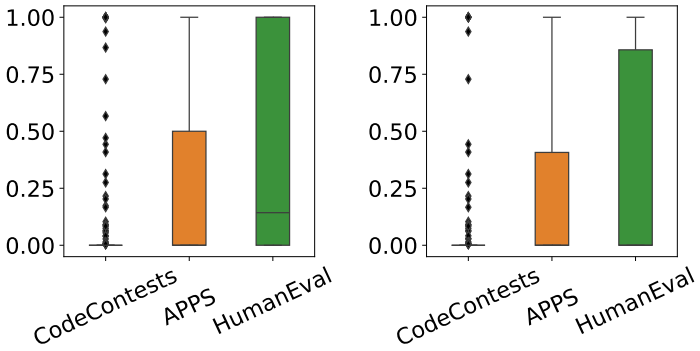
4.1.1 RQ1.1: Semantic Similarity. Semantic similarity is measured by the following metrics: test pass rate and OER (output equivalence rate), and OER excluding exceptions. As mentioned in Section 3.4, each coding problem has five test pass rates, we use the variance and maximum difference of these five values to indicate ChatGPT's non-determinism in generating code for the task. We also report the mean value, which represents the average correctness of the generated code. For OER or OER (no ex.), we compare the equivalence across all the five code candidates as well as between every two candidates. For each dataset, we report the distribution of different measurements in Figure 2 and Figure 3. The mean measurement values for all the coding problems (the mean value inside each bar in each bar chart) in a dataset are shown in Table 2. The max diff refers to the maximum value of the max diff among all the coding problems. In addition, Table 2 also shows the "Ratio of worst cases", which is the ratio of problems with maximum diff of test pass rate being 1 or OER being 0.



(a) Variance (b) Mean (c) Max Diff

Fig. 2. RQ1.1: Distribution of semantic similarity in terms of test pass rate.

From Figure 2, Figure 3, and Table 2, we observe that ChatGPT is very unstable in generating semantically consistent code candidates. In particular, the ratios of tasks with zero equal test output (i.e., OER=0) among the five code candidates are 75.76%, 51.00%, and 47.56% for the three datasets, respectively. This indicates that for the majority of the cases, ChatGPT generates code candidates with completely different semantics from identical instructions.



(a) OER (b) OER (no ex.)

Fig. 3. RQ1.1: Distribution of semantic similarity in terms of test output equivalence rate (OER and OER (no ex.)).

Table 2. RQ1.1: Results of semantic similarity. OER and OER (no ex.) are the output equivalence rate and the equivalence rate excluding exceptions.

Semantic similarity	Metric	CodeContests	APPS	HumanEval
Test pass rate	Mean value	0.16	0.42	0.63
	Mean variance	0.03	0.04	0.09
	Mean max diff	0.24	0.35	0.53
	Max diff	1.00	1.00	1.00
	Ratio of worst cases	3.64%	10.40%	39.63%
OER	Mean value	0.09	0.27	0.39
	Pair mean value	0.27	0.47	0.67
	Worst value	0.00	0.00	0.00
	Ratio of worst cases	75.76%	51.00%	47.56%
OER (no ex.)	Mean value	0.06	0.25	0.35
	Pair mean value	0.19	0.42	0.61
	Worst value	0.00	0.00	0.00
	Ratio of worst cases	81.21%	53.40%	51.22%

The mean variance of the test pass rate is relatively small from Table 2, ranging between 0.03 and 0.09, this is because the test pass rate of different code candidates is often equally worse, as can be observed from Figure 2.(a). However, the max diff of the test pass rate reaches 1.00 for all three datasets and accounts for 39.63% of the problems in HumanEval, the most widely used code generation benchmark. This indicates the correctness of code candidates generated from the same instruction can vary significantly. The large difference in different datasets also sheds light on the importance of using multiple datasets when assessing the code generation performance for large language models.

Our statistical analysis with Kruskal-Wallis test shows that, in 92.1% of CodeContests, 39.4% of APPS, and 40% of HumanEval, the outputs of the code are indeed significantly different, where the p-value under the Kruskal-Wallis test is less than 0.05.

Answer to RQ1.1: The semantic difference among the code generated by ChatGPT in different requests is significant. In particular, the ratio of coding tasks with not a single equal test output among the five different requests is 75.76%, 51.00%, and 47.56% for CodeContests, APPS, and HumanEval, respectively. In addition, the maximum difference of the test pass rate reaches 1.00 for all three datasets and accounts for 39.63% of the problems in HumanEval, the most widely used code generation benchmark.

4.1.2 RQ1.2: Syntactic Similarity. Syntactic similarity measures the text similarity among code candidates. In our experiment, the syntactic similarity is evaluated by the following metrics: LCS and LED (more details in Section 3.4). For the five code candidates for each coding problem, we use the first code candidate as a reference and calculate the LCS and LED between the reference and the remaining four candidates. In addition, we calculate LCS and LED with code candidates in pairs, for each pair combination. Thus, each problem has four LCS values and LED values, and 20 LCS and LED values in pairs, each value indicating a syntactic similarity. We use the mean of these four values as well as the worst of them (i.e., the smallest value for LCS and the largest value for LED), and the mean of these 20 values calculated in pairs to represent each problem's syntactic similarity. Figure 4 shows the distribution of LCS and LED for all the problems in each dataset. Table 3 shows the mean, mean worst, and pair mean LCS and LED values for all the coding problems (the mean value inside each bar in the figures) in a dataset.

Table 3. RQ1.2: Syntactic similarity. Lower LCS and higher LED indicate lower syntactic similarity.

Syntactic Similarity	Metric	CodeContests	APPS	HumanEval
LCS	Mean value	0.22	0.23	0.42
	Mean worst value	0.16	0.16	0.25
	Pair mean value	0.23	0.24	0.41
LED	Mean value	58.80	47.37	26.56
	Mean worst value	77.46	61.55	43.91
	Pair mean value	58.86	46.94	27.10

We observe that the code candidates generated from the same instruction also differ largely in the syntactic measure. Specifically, the mean LCS is 0.22, 0.23, and 0.42 for CodeContests, APPS, and HumanEval, respectively, indicating the mean ratio of the longest common subsequences among the code candidates.

For the three datasets, we could see from Table 3 that the lowest LCS and largest LED values both happen for the CodeContests dataset. By contrast, the largest LCS and smallest LED values both happen for HumanEval. This indicates that ChatGPT is most unstable syntactically for the code generation tasks in CodeContests, and most stable for HumanEval. We further explore the correlation between different similarities and code task features in Section 4.4.

Answer to RQ1.2: Code candidates generated by ChatGPT in different requests also differ significantly in syntax. The mean syntax similarity (LCS) is only 0.22, 0.23, and 0.42 for CodeContests, APPS, and HumanEval, respectively.

4.1.3 RQ1.3: Structural Similarity. Structural similarity measures the codes' similarity based on their AST. In our experiment, the structural similarity is mainly measured by the tool `pycode_similar` with two different settings, namely United_Diff and Tree_Diff (more details in Section 3.4). For the five code candidates for each coding problem, we use the first code candidate as a reference and calculate the structural similarity between the first

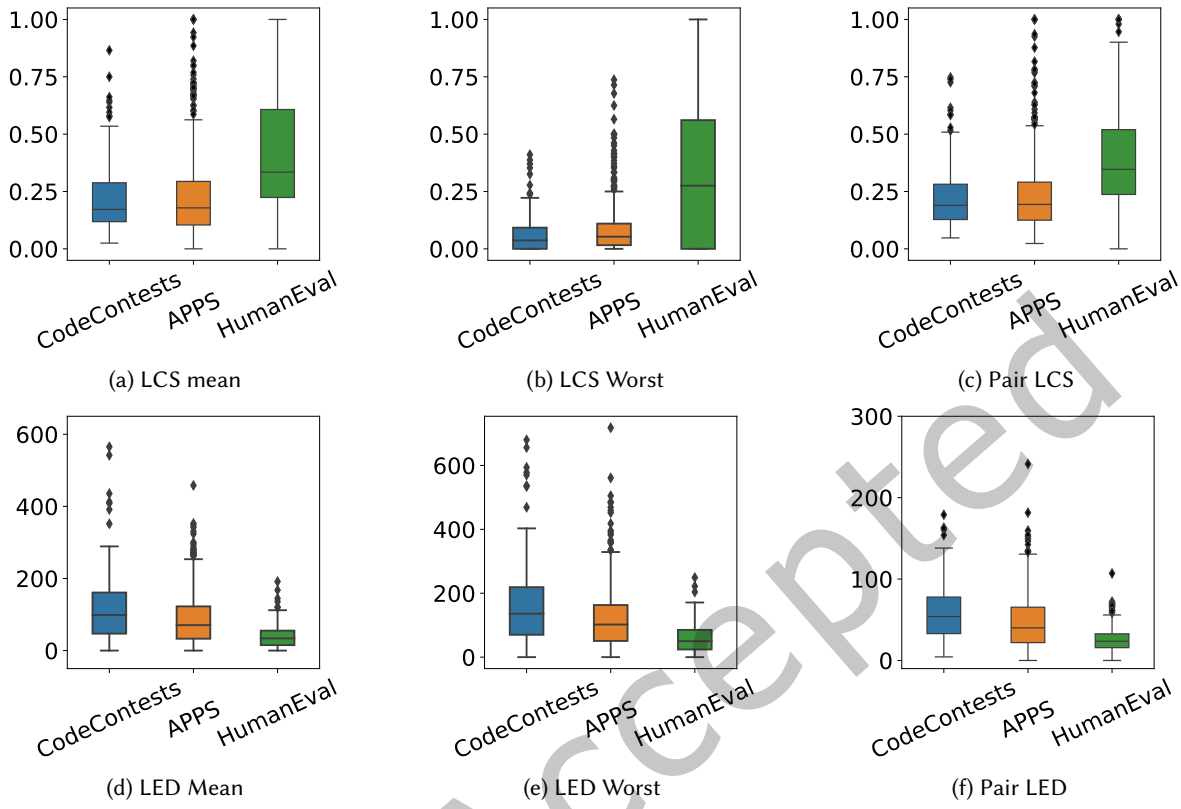
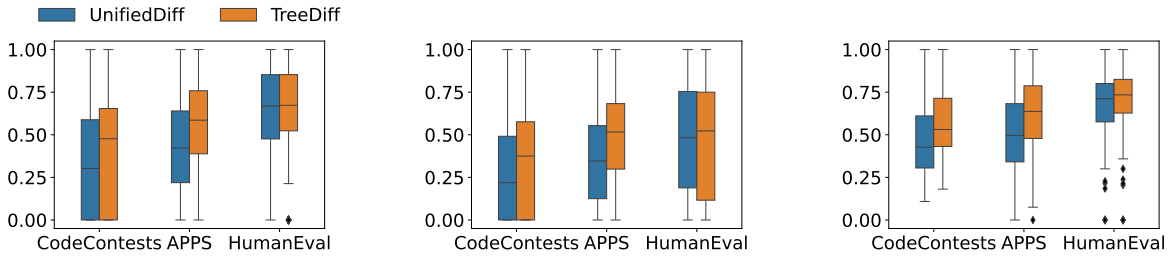


Fig. 4. RQ1.2: Distribution of syntactic similarity (LCS & LED). Lower LCS and higher LED indicate less syntactic similarity.

candidate with the remaining four candidates under United_Diff and Tree_Diff settings. We also calculate the structural similarity with code candidates in pairs, with a total of 20 pair mean values. Thus, each problem has four mean values and 20 pair mean values for United_Diff and Tree_Diff respectively, with each value indicating a structural similarity measure. We use the mean of these four values, the worst of them, and their pair mean values (i.e., the smallest value for United_Diff and Tree_Diff) to represent each problem's structural similarity. Fig 5 shows the distribution of United_Diff and Tree_Diff for all the problems in each dataset. Table 4 shows the mean, mean worst values, and pair mean values under United_Diff and Tree_Diff settings for all the coding problems (the mean value inside each bar in the figures) in a dataset.

Table 4. RQ1.3: Structural similarity.

Structural Similarity	Metric	CodeContests	APPS	HumanEval
United_Diff	Mean value	0.33	0.43	0.60
	Mean worst value	0.27	0.35	0.47
	Pair mean value	0.46	0.52	0.67
Tree_Diff	Mean value	0.41	0.54	0.62
	Mean worst value	0.33	0.47	0.48
	Pair mean value	0.56	0.63	0.70



(a) Mean (b) Mean Worst (c) Pair Mean

Fig. 5. RQ1.3: Structural Similarity (United_Diff & Tree_Diff).

We observe that the code candidates generated from the same instruction show great similarity in structure. Specifically, the mean values are 0.33, 0.43, and 0.60 under the United_Diff setting, and 0.41, 0.54, and 0.62 under Tree_Diff setting for CodeContests, APPS, and HumanEval, respectively.

For the three datasets, we could see from Table 4 that the lowest values under United_Diff and Tree_Diff happen for the CodeContests dataset. By contrast, the largest values under the two settings both happen for HumanEval. This indicates that ChatGPT is most unstable in structure for the code generation tasks in CodeContests, and most stable for HumanEval. We further explore the correlation between different similarities and task features in RQ4.

Answer to RQ1.3: Code candidates show high structural similarity under UnitedDiff and TreeDiff settings. We observe that the code candidates generated from the same instruction have high similarity in structure. Specifically, the mean values are 0.33, 0.43, and 0.60 under the United_Diff setting, and 0.41, 0.54, and 0.62 under Tree_Diff setting for CodeContests, APPS, and HumanEval, respectively.

4.2 RQ2: Influence of Temperature

The default temperature of ChatGPT is 1[11]. This RQ explores whether the code generation non-determinism of ChatGPT changes with the temperature changes. We use identical measurements as in RQ1. We show our experiment results on CodeContests only. Results for other datasets are on our homepage [3].

Table 5 shows the results. Overall, we observe that when *temperature*=0, ChatGPT has better determinism than the default configuration (*temperature*=1) for all three types of similarities. However, setting the temperature to 0 does not completely avoid non-determinism. Take OER as an example, there are still 43.64% (CodeContests), 27.40% (APPS), and 18.29% (HumanEval) of problems with no equal test output among the five code candidates. This is contrary to many people's belief that setting the temperature to 0 can make ChatGPT deterministic [7, 13, 39], because when setting the temperature to 0, the model applies greedy sampling which should indicate full determinism, with the logit value for the next token being a pure function of the input sequence and the model weights. The reason for such non-determinism with the temperature being zero is still controversial [1], with different hypotheses such as floating point, unreliable GPU calculations, and its sparse MoE architecture failing to enforce per-sequence determinism [33, 55]. The details for all the non-deterministic coding tasks and their test outputs with temperature=0 are on our homepage [3].

[11] https://platform.openai.com/docs/api-reference/chat/create#chat-create-temperature

Table 5. RQ2: Influence of temperature (CodeContests).

Temperature	Test Pass Rate					
	Mean value	Mean variance	Mean max diff	Max diff	Ratio of worst cases	
0	0.15	0.01	0.11	1.00	1.82%	
0.5	0.16	0.02	0.15	1.00	2.42%	
1	0.16	0.03	0.24	1.00	3.64%	
Temperature	OER			OER (no ex.)		
	Mean value	Ratio of worst cases	Pair mean value	Mean value	Ratio of worst cases	Pair mean value
0	0.37	43.64%	0.59	0.27	54.55%	0.46
0.5	0.18	62.42%	0.37	0.13	68.48%	0.28
1	0.09	75.76%	0.27	0.06	81.21%	0.19
Temperature	LCS			LED		
	Mean value	Mean worst value	Pair mean value	Mean value	Mean worst value	Pair mean value
0	0.61	0.44	0.62	23.45	35.87	22.31
0.5	0.33	0.23	0.34	44.48	62.02	44.89
1	0.22	0.16	0.23	58.80	77.46	58.86
Temperature	United_Diff			Tree_Diff		
	Mean value	Mean worst value	Pair mean value	Mean value	Mean worst value	Pair mean value
0	0.41	0.39	0.67	0.50	0.46	0.74
0.5	0.61	0.49	0.63	0.69	0.58	0.71
1	0.33	0.27	0.46	0.41	0.33	0.56

When *temperature*=0.5, we observe that ChatGPT tends to generate code candidates that are more deterministic than temperature=1, but less deterministic than temperature=0. This is as expected because the higher temperature brings more creativity to ChatGPT and affects its ability to generate similar code (as can be observed from the other measurements, such as LCS and LED). Nevertheless, we observe that the value of test pass rates among the three different temperatures are similar, which indicates that low temperature might be a better choice given the comparable test pass rate and the low degree of non-determinism.

Answer to RQ2: Contrary to the widely held belief (and common practices), setting the temperature to 0 does not guarantee determinism in code generation, although it indeed brings more determinism than the default configuration (*temperature*=1) for all three types of similarities. We also observe that the values of test pass rate among the three different temperatures are similar, indicating that low temperature might be a better choice for code generation tasks.

4.3 RQ3: Non-determinism Comparison with Top Candidates in the Same Prediction

RQ1 and RQ2 compare the similarity of 5 code candidates generated in multiple requests. Each candidate is the top candidate in each request. However, ChatGPT can also generate 5 code candidates within the same request (the top 5 candidates ranked by their predictive probabilities). This RQ compares the non-determinism degree of code candidates for the two request configurations mentioned above (with temperature = 1 and temperature =0). Table 6 shows the results for CodeContests, the results for the two other datasets are on our homepage [3]. For ease of presentation, we use R1 to refer to one-time requests, and R2 to refer to multiple requests.

Our results reveal that when setting temperature=1, it is difficult to tell which way of requesting is more deterministic. For semantic similarity, R1 and R2's performance are similar among three datasets. Code candidates requested in R1 are slightly more random than those requested in R2 in terms of syntactic similarity since those

requested in R1 have lower LCS values and higher LED values. However, code candidates requested in R1 are slightly more stable than those requested in R2 when it comes to similarity because those requested in R1 have higher structural similarity values in both United_Diff and Tree_Diff settings.

When temperature is 0, the difference between the two request ways is obvious. Code Candidates requested by R1 show higher determinism than those requested by R2. When requesting by R1, the ratio of worst cases, where max diff is close to 0 (1.20%), and the OER and OER (no ex.) are higher than R2 and close to 1. The LCS values are higher than the values under other temperatures and LED values are lower than the values under other temperatures, which indicates higher determinism. Among the three datasets, the structural similarity values are also higher than the values in other temperatures, which means the code candidates are more close to each other in terms of their AST structure.

Table 6. RQ3: Similarity for different request ways (CodeContests), where t represents the temperature setting.

Request Way	Test Pass Rate					
	Mean value	Mean variance	Mean max diff	Max diff	Ratio of worst cases	
R1 (t=1)	0.17	0.03	0.28	1.00	8.70%	
R2 (t=1)	0.16	0.03	0.24	1.00	3.64%	
R1 (t=0)	0.18	0.00	0.00	0.00	1.20%	
R2 (t=0)	0.15	0.01	0.11	1.00	1.82%	
Request	OER			OER (no ex.)		
	Mean value	Ratio of worst cases	Pair mean value	Mean value	Ratio of worst cases	Pair mean value
R1 (t=1)	0.09	76.09%	0.27	0.04	83.70%	0.18
R2 (t=1)	0.09	75.76%	0.27	0.06	81.21%	0.19
R1 (t=0)	1.00	1.20%	1.00	0.81	12.05%	0.81
R2 (t=0)	0.37	43.64%	0.59	0.27	54.55%	0.46
Request	LCS			LED		
Way	Mean value	Mean worst value	Pair mean value	Mean value	Mean worst value	Pair mean value
R1 (t=1)	0.21	0.15	0.20	61.30	82.73	63.09
R2 (t=1)	0.22	0.16	0.23	58.80	77.46	58.86
R1 (t=0)	1.00	1.00	1.00	0.00	0.00	0.00
R2 (t=0)	0.61	0.44	0.62	23.45	35.87	22.31
Request	United_Diff			Tree_Diff		
Way	Mean value	Mean worst value	Pair mean value	Mean value	Mean worst value	Pair mean value
R1 (t=1)	0.98	0.98	0.98	0.98	0.98	0.98
R2 (t=1)	0.33	0.27	0.46	0.41	0.33	0.56
R1 (t=0)	1.00	1.00	1.00	1.00	1.00	1.00
R2 (t=0)	0.41	0.39	0.67	0.50	0.46	0.74

Answer to RQ3: Under default temperature, the top-5 code candidates from one single request have similar non-determinism with the 5 top-1 candidates from different requests for ChatGPT when the temperature is 1 (default temperature of ChatGPT), but higher determinism when the temperature is 0.

4.4 RQ4: Coding Tasks Features and Non-determinism Degree

Our previous experiments demonstrate that there are many non-determinisms in ChatGPT in code generation. This RQ investigates what affects such non-determinism by checking the correlation between characteristics of coding tasks and similarity metric values. We use three datasets for this RQ. For all the datasets, we consider *description length* as one of their extrinsic features. Because only the CodeContests dataset has various extrinsic

features for each coding task, including *difficulty*, *time limit*, and *CF rating*, we consider these features as extrinsic features for the CodeContest dataset as well. Although APPS does have *difficulty* features, the *difficulty* features in APPS are shown as categories, namely, 'introductory', 'interview', and 'competition', which makes it hard to map them into numerical values. Therefore, our experiment does not include *difficulty* as an extrinsic feature for the APPS dataset.

In CodeContests, the *CF rating* of a problem is a quantitative measure that represents the problem's relative difficulty level compared to other problems on the Codeforces platform. The *difficulty* of a problem is a qualitative measure that indicates the problem's level of complexity and the programming knowledge and skills required to solve it. The *timeout* indicates the program's maximum running time limitation. In addition, we also consider description length (i.e., number of characters) for each coding task. Note that in this section, we only focus on correlation analysis, and we do not aim to obtain any causal conclusions.

Figure 6 shows the results for code problems in CodeContests under temperature=1. The rest figures can be found on our homepage [3]. We observe that description length has a negative correlation with most of the measurements, except LED. This means that problems with longer descriptions tend to generate code with more randomness. We suspect that this is because a longer description may reduce ChatGPT's understanding of the coding requirements. With longer descriptions, different code candidates tend to be uniformly worse in their pass rates. Moreover, the description length has a negative correlation with LCS and structural measurements and a positive correlation with LED, which means that problems with longer descriptions tend to yield more inconsistent code candidates in syntax and structure. For temperature = 0, we observe that description length still has a negative correlation with most of the measurements, except LED, which is similar to the correlation result under temperature=1.

The *difficulty* has a positive correlation with the LED and a negative correlation with LCS, which means that the problem with a higher *difficulty* level has high non-determinism in syntax. Similar to *difficulty*, *CF rating* also has a positive correlation with the LED and a negative correlation with LCS.

In the following, we provide some specific examples to further illustrate our observations above. In exploring the relationship between the length of a code problem description and the degree of non-determinism, two contrasting examples in the CodeContests dataset corroborate our findings. The first example, '1599_E. Two Arrays', with a description length of 2149, show a pattern that code generation with a longer description code problem has a higher degree of non-determinism. Below is the description of the first code problem, where we present only the core part of the description due to the extensive length of the overall content.

1599_E. Two Arrays
You are given two integer arrays of length N, A1, and A2. You are also given Q queries of 4 types:
1 k l r x : $setAk_i := min(Ak_i, x)$ for each $l \leq i \leq r$.
2 k l r x : $setAk_i := max(Ak_i, x)$ for each $l \leq i \leq r$.
3 k l r x : $setAk_i := Ak_i + x$ for each $l \leq i \leq r$.
4 l r : $findthe(\sum_{i=l}^{r} F(A1_i + A2_i))\%(10^9 + 7)$
where $F(k)$ is the k-th Fibonacci number $(F(0) = 0, F(1) = 1, F(k) = F(k-1) + F(k-2))$, and $x\%y$ denotes the remainder of the division of x by y. You should process these queries and answer each query of the fourth type.

This problem exhibits high non-determinism, as indicated by its measurement results across multiple tests (i.e., the test case rate variance is 0.13, the OER value is zero, the LCS mean value is 0.15, the mean LED value is 111.5, and both the United_Diff and Tree_Diff values are zero), suggesting a rather high fluctuation. The detailed description potentially covers a wide array of scenarios, which may distract the attention from LLMs, which results in inconsistent test results and higher non-determinism.

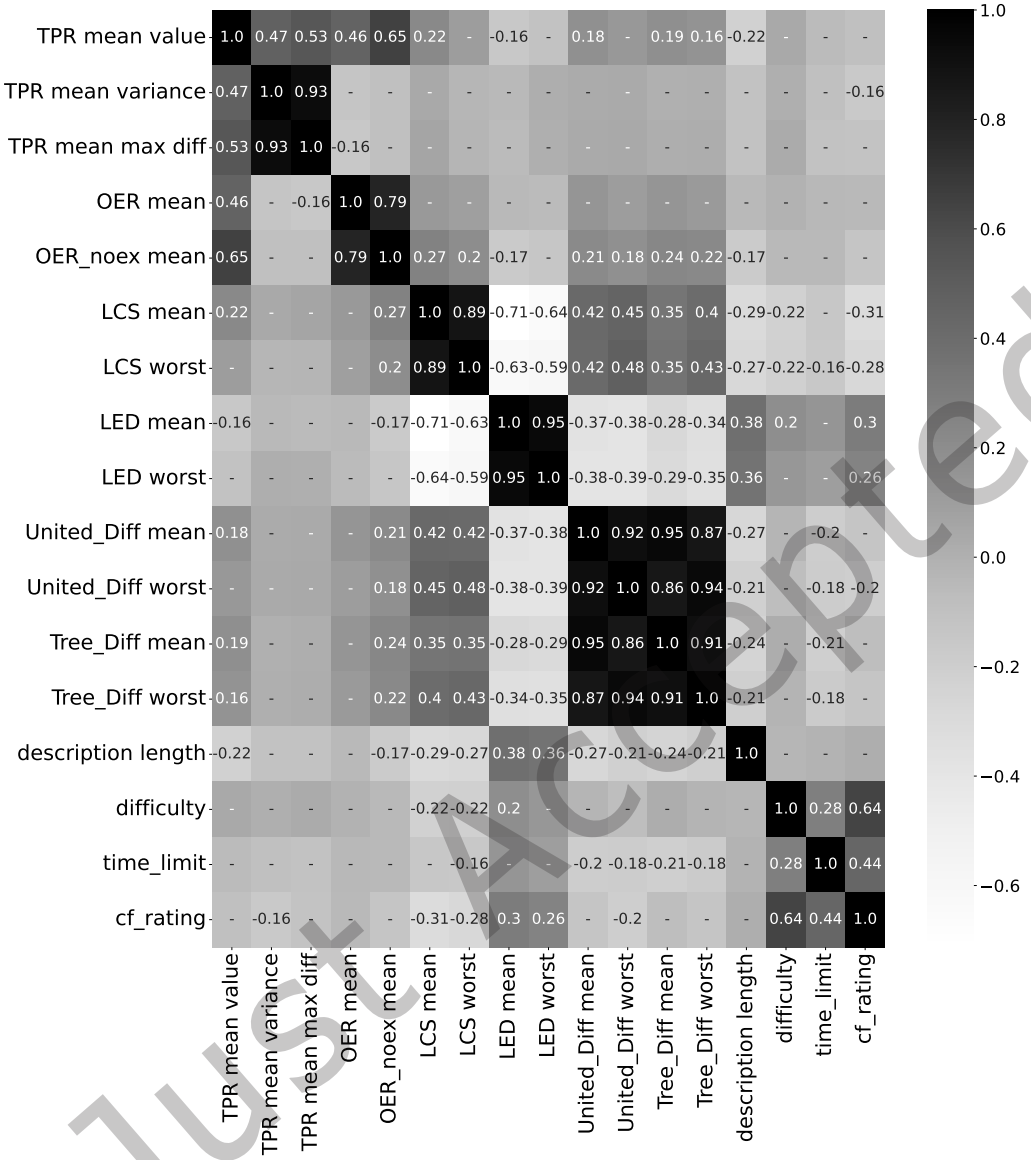


Fig. 6. RQ4: Correlations between coding tasks and non-determinism (CodeContests, temperature=1). Only significant correlations will be displayed on the heatmap, while the insignificant correlations (i.e. p-value > 0.05) are masked by '-'.

The second example '1575_M. Managing Telephone Poles', with a description length of 1511, shows a pattern that a shorter description leads to more stability in code generation. Below is the description of the second code problem, where we present only the core part of the description due to the extensive length of the overall content.

1575_M. Managing Telephone Poles
Mr. Chanek's city can be represented as a plane. He wants to build a housing complex in the city. There are some telephone poles on the plane, which is represented by a grid of size $(n+1)(m+1)$.
There is a telephone pole at (x, y) if $a_{x,y} = 1$. For each point (x, y), define $S(x, y)$ as the square of the Euclidean distance between the nearest pole and (x, y).
Formally, the square of the Euclidean distance between two points (x_1, y_1) and (x_2, y_2) is $(x_2 - x_1)^2 + (y_2 - y_1)^2$.
To optimize the building plan, the project supervisor asks you the sum of all $S(x, y)$ for each $0 \leq x \leq n$ and $0 \leq y \leq m$. Help him by finding the value of $\sum_{x=0}^{n} \sum_{y=0}^{m} S(x, y)$.

The test pass rates are consistently 1.0 across all tests, with a variance of 0.0, showing no deviation in the generated code candidates. The LCS mean value is 0.74, and the LED mean value is 3.5, which indicates a high syntactical stability. Structural similarity is 0.21 and 0.38 under United_Diff and Tree_Diff settings, which shows the code candidates still vary in their AST. Here, the shorter description does not introduce ambiguity but rather lets ChatGPT focus on critical details, leading to a uniform understanding of the code problem and better generation performance.

Answer to RQ4: A coding task with a longer description and higher difficulty tends to suffer from more non-determinism in the generated code in terms of code syntax and structure. The generated code also tends to be more buggy.

4.5 RQ5: GPT-4 vs. GPT-3.5

GPT-4 is believed to be "more reliable, creative, and able to handle much more nuanced instructions than GPT-3.5" [52]. This research question compares GPT-3.5 and GPT-4 in the non-determinism degree of code generation. To answer this research question, we keep the default setting and use all the measurements listed in RQ1. In this paper we report the results only on the CodeContests dataset (with temperature=1). For the results in the other two datasets, we list them on our homepage [3].

For temperature=1, we can observe that GPT-4 is slightly more deterministic than GPT-3.5, with lower test pass rate variance, lower ratio of worst cases, lower OER and OER (no ex.), lower LCS, higher LED, and lower structural similarity under two settings. However, for temperature=0, the analysis, as evidenced by the results in tables comparing GPT-4 across CodeContests, APPS, and HumanEval datasets, demonstrates that GPT-4's non-determinism is pronounced and largely parallels that of GPT-3.5. Across these datasets, similarity metrics indicate comparable levels of non-determinism across three different evaluation methods.

Answer to RQ5: The non-determinism issue of GPT-4 is lightly less severe than GPT-3.5 under temperature=1, while the non-determinism issue of GPT-4 is similar to GPT-3.5 under temperature=0.

4.6 RQ6: Influence of Prompt Engineering Strategies on the Non-determinism

This research question explores how different prompt engineering strategies influence the degree of non-determinism in code generation. We design two extra prompts in addition to the default one used for previous RQs. The first prompt is "Generate Python3 code (Markdown), make the code as concise as possible". This prompt aims to lead ChatGPT to generate short and concise programs, which may make the results more deterministic. The second prompt is "Generate Chain-of-Thought steps of how to solve the problem first, and then generate Python3 code (Markdown)", thereby demanding an initial conceptual explanation followed by the code. Then,

Table 7. RQ5: Non-determinism of GPT-4 v.s. GPT-3.5 (CodeContests)

Model	Test Pass Rate					
	Mean value	Mean variance	Mean max diff	Max diff	Ratio of worst cases	
GPT-4 (t=1)	0.14	0.01	0.09	1.00	1.21%	
GPT-3.5 (t=1)	0.16	0.03	0.24	1.00	3.64%	
GPT-4 (t=0)	0.14	0.01	0.08	1.00	1.21%	
GPT-3.5 (t=0)	0.15	0.01	0.11	1.00	1.82%	
Model	OER			OER (no ex.)		
	Mean value	Ratio of worst cases	Pair mean value	Mean value	Ratio of worst cases	Pair mean value
GPT-4 (t=1)	0.35	46.06%	0.58	0.25	55.76%	0.46
GPT-3.5 (t=1)	0.09	75.76%	0.27	0.06	81.21%	0.19
GPT-4 (t=0)	0.37	41.21%	0.59	0.27	52.73%	0.46
GPT-3.5 (t=0)	0.37	43.64%	0.59	0.27	54.55%	0.46
Model	LCS			LED		
	Mean value	Mean worst value	Pair mean value	Mean value	Mean worst value	Pair mean value
GPT-4 (t=1)	0.61	0.45	0.62	24.54	39.74	24.81
GPT-3.5 (t=1)	0.22	0.16	0.23	58.80	77.46	58.86
GPT-4 (t=0)	0.61	0.44	0.61	24.45	40.14	24.12
GPT-3.5 (t=0)	0.61	0.44	0.62	23.45	35.87	22.31
Model	United_Diff			Tree_Diff		
	Mean value	Mean worst value	Pair mean value	Mean value	Mean worst value	Pair mean value
GPT-4 (t=1)	0.78	0.68	0.79	0.82	0.74	0.84
GPT-3.5 (t=1)	0.33	0.27	0.46	0.41	0.33	0.56
GPT-4 (t=0)	0.78	0.68	0.79	0.83	0.75	0.84
GPT-3.5 (t=0)	0.41	0.39	0.67	0.50	0.46	0.74

each prompt is followed by the code problem description. In the following, we use 'Concise prompt' to refer to the first prompt engineering strategy, and use 'CoT prompt' to refer to the second one for short.

The results in Table 8 show that for temperature=1, the difference of non-determinism between different prompt engineering techniques is not very obvious in the three datasets. With more instruction information provided in the prompt, Concise and CoT prompts have similar performance with each other. However, under temperature=0, in CodeContests, requests with CoT prompt show high mean test pass rates but this kind of prompt suffers from high randomness. Compared with the Base prompt and Concise prompt, the CoT prompt has a higher mean-variance (0.02), higher mean maximum difference (0.15), and a rather higher ratio of worst cases (1.82%). Also, the results in OER and OER (no ex.) show that CoT's mean value of OER and OER (no ex.) are lower than Base and Concise, which can also be told from the high ratio of worst cases in both OER and OER (no ex.) with 46.06% and 54.55%. Opposite from CoT, code candidates generated from Concise prompt are more semantically deterministic. Code candidates generated by the CoT prompt have a low mean LCS value (0.38) and high LED value (39.31), while those generated from the Concise prompt have a high mean LCS value (0.07) and low LED value (11.77). The other measurements in LCS and LED also support the above phenomenon. When it comes to structural similarity, under two different measurement settings, code candidates generated from the CoT prompt have significantly higher randomness than the code generated from Concise prompt. Our experiment results show a similar situation in both APPS and HumanEval, where code generated from the Concise prompt ends up way more deterministic than code generated from the CoT prompt.

Table 8. RQ6: Prompt engineering techniques (CodeContests), where t refers to temperature.

Prompt	Test Pass Rate					
	Mean value	Mean variance	Mean max diff	Max diff	Ratio of worst cases	
Concise (t=1)	0.15	0.02	0.19	1.00	3.64%	
Base (t=1)	0.16	0.03	0.24	1.00	3.64%	
CoT (t=1)	0.15	0.02	0.19	1.00	3.64%	
Concise (t=0)	0.16	0.01	0.10	1.00	0.61%	
Base (t=0)	0.15	0.01	0.11	1.00	1.82%	
CoT (t=0)	0.19	0.02	0.15	1.00	1.82%	
Prompt	OER			OER (no ex.)		
	Mean value	Ratio of worst cases	Pair mean value	Mean value	Ratio of worst cases	Pair mean value
Concise (t=1)	0.10	76.36%	0.26	0.06	81.82%	0.17
Base (t=1)	0.09	75.76%	0.27	0.06	81.21%	0.19
CoT (t=1)	0.10	73.94%	0.26	0.08	80.0%	0.19
Concise (t=0)	0.39	41.82%	0.63	0.31	49.09%	0.54
Base (t=0)	0.37	43.64%	0.59	0.27	54.55%	0.46
CoT (t=0)	0.28	46.06%	0.50	0.19	54.55%	0.36
Prompt	LCS			LED		
	Mean value	Mean worst value	Pair mean value	Mean value	Mean worst value	Pair mean value
Concise (t=1)	0.22	0.16	0.22	61.53	83.01	62.52
Base (t=1)	0.22	0.16	0.23	58.80	77.46	58.86
CoT (t=1)	0.23	0.15	0.23	59.55	77.68	57.05
Concise (t=0)	0.70	0.53	0.71	11.77	20.55	12.14
Base (t=0)	0.61	0.44	0.62	23.45	35.87	22.31
CoT (t=0)	0.38	0.24	0.39	39.31	58.28	39.81
Prompt	United_Diff			Tree_Diff		
	Mean value	Mean worst value	Pair mean value	Mean value	Mean worst value	Pair mean value
Concise (t=1)	0.44	0.34	0.48	0.54	0.42	0.59
Base (t=1)	0.33	0.27	0.46	0.41	0.33	0.56
CoT (t=1)	0.45	0.35	0.51	0.55	0.43	0.61
Concise (t=0)	0.83	0.74	0.84	0.88	0.82	0.89
Base (t=0)	0.41	0.39	0.67	0.50	0.46	0.74
CoT (t=0)	0.71	0.58	0.72	0.78	0.67	0.79

Answer to RQ6: Under temperature=1, the difference in non-determinism among different prompt engineering techniques is not obvious. When setting temperature=0, the code candidates generated from the Concise prompt are more deterministic than our Base prompt, while those code candidates generated from the CoT prompt suffer from higher randomness than our Base prompt.

5 THREATS TO VALIDITY

The threats to *internal* validity mainly lie in the implementation of our experiment and result analysis. To reduce the first threat, we checked our code twice, once during the experiment stage, and once during the record analysis stage. To reduce the second threat, the two authors independently analyzed the experiment results and drew experimental conclusions separately. Once their analysis results were different, the third author discussed with them to determine the final result.

The threats to *external* validity mainly lie in the datasets, GPT versions, and prompt design in our study. To reduce the threat in datasets, we use three diverse datasets that are widely used in code generation tasks. Additionally, the problems in our dataset are from different contests with different difficulties. For example,

CodeContests is the most challenging dataset, while HumanEval is the easiest, in terms of the average difficulty of coding problems. To reduce the threat in GPT versions, we consider the two newest versions of GPT: GPT-3.5 and GPT-4, and compare their non-determinism from multiple aspects. To reduce the threat of prompt design, we use the most typical prompts that are the most widely used in LLM-based code generation and design an RQ to study their influence on non-determinism.

Another primary concern highlighted in our analysis revolves around the operationalization of semantic, syntactic, and structural similarities into measurable metrics for assessing code similarity. The approach of measuring semantic similarity through the comparison of test execution outputs, while practical, presents a notable limitation. It potentially oversimplifies the multifaceted nature of semantic similarity, which should ideally encapsulate the code's meaning and functionality rather than merely its output. This method risks ignoring the intricate logic and diverse correct solutions that different pieces of code may offer. To reduce the threat in measurement tools, we consider three types of similarities and choose at least two measurements for each type of similarity, and we also apply statistical analysis techniques to enhance our experiment results. For the HumanEval dataset, we evaluate our measurement on an external testset, EvalPlus [42]. The result shows that our measurements show similar evaluation results, which supports the robustness of our chosen measurements.

However, it is important to acknowledge certain *limitations* within our study that may affect the breadth of its applicability and the generalizability of its findings. Firstly, our analysis does not extend to the impact that different programming languages might have on the non-determinism of code generation. Programming languages vary widely in syntax, semantics, and complexity, which can influence how LLMs like ChatGPT interpret and generate code, potentially affecting the degree of non-determinism in the output. Secondly, our work only adopts a few methods for measuring code similarity. There is no unified standard for measuring code similarity. It is challenging to cover all the code similarity measurements. Other methods include embedding-based similarity measure methods, using pre-trained code language models, such as CodeBERT [17] and GraphCodeBERT [22]. Thirdly, the influence of the prompt on non-determinism is not fully considered. The specificity, clarity, and technical depth of prompts provided to ChatGPT can significantly influence the model's output, suggesting that prompts could be a crucial factor in understanding non-determinism. Fourthly, our study focuses exclusively on ChatGPT. While ChatGPT is a prominent LLM used for code generation, it is not the only one. The landscape of LLMs is diverse, with models trained on different datasets, architectures, and objectives. Therefore, our findings may not apply to other LLMs used for similar purposes.

6 RELATED WORK

6.1 Code Generation

Code generation generates programs that need to satisfy all the constraints defined by the underlying task. Usually, the constraints are represented in various forms, e.g. input/output pairs, examples, problem descriptions, partial programs, and assertions. Relatively early work includes deductive synthesis approaches [19, 46] and inductive synthesis approaches [8, 57, 58, 60]. The deductive synthesis approach operated under the assumption that a comprehensive and precise formal specification of the user's desired intention would be provided. However, in many instances, this turned out to be just as intricate and challenging as creating the actual program. While the inductive synthesis approach was based on inductive specifications such as input/output pairs and examples etc, such as works on Lisp programs [8, 57, 60], Pygmalion [58] and more recently FlashFill [20]. More information could be found in a survey [21], which covers notable work on the development of program synthesis approaches.

In recent years, more and more researchers apply neural networks in code generation. Yin and Neubig [70] combine the grammar rules with the decoder and propose a syntax-driven neural architecture to improve code generation performance. Instead of RNN, Sun et al. [61] propose a grammar-based structural CNN to capture the long dependency in code. Bolin et al. [66] propose a dual learning framework that jointly trains the code

generation model and code summarization model together to achieve better performance in both tasks. Xu et al.[68] present a user study in-IDE code generation, demonstrating challenges such as time efficiency, correctness, and code quality, as well as the willingness to use code generation tools from developers.

6.2 Language Model for Code generation

The triumph of transformers in natural language modeling [9] has stimulated considerable interest among researchers in applying transformer models for code generation. Existing research on code generation models can be classified into three categories: sequence-based techniques, tree-based methods, and pre-trained models.

Sequence-based techniques take code as a sequence of tokens and employ language models to produce source code one token at a time based on input descriptions. Ling et al. [40] propose a generative model for code generation along with a character level softmax and multi-pointer network to address the problem of generating code from a mixed language and structured specification, and receiving success in trading card games (Magic the Gathering and Hearthstone). Hashimoto et al. [24] train a retrieval model with a noisy encoder-decoder to enable similar code retrieving, and then use the similar code as an additional input to improve the performance of the generator.

Tree-based methods generate a parse tree of the code, e.g. Abstract Syntax Tree (AST), based on the input description, and then convert the parse tree into the corresponding code. Dong et al. [14] encode natural language utterances into vectors and generate their corresponding logical forms as trees using the LSTM model. Yin et al. [71] propose a semantic parser 'Tranx', which generates the tree-construction action sequence with a transition-based neural model, and constructs the AST from the action sequence.

Pre-trained models are obtained from training on massive data of source code, which could be later fine-tuned on certain datasets for code generation purposes. Encoder pre-trained models, such as CodeBERT [17], usually are trained with two objectives, i.e., Masked Language Modeling and Replaced Token Detection. During the fine-tuning phase, the input should be fed in the same way as the pre-training phrase, so that semantic relevance could be measured. Decoder pre-trained models are designed to predict the next token based on a given input context. GPT-series [56] are typical Decoder pre-trained models, and based on GPT, there are many efforts on code generation. Based on GPT-2, Lu et al. [45] provide CodeGPT for code completion and text-to-code generation. After GPT-3 was developed, CodeX[12] and GitHub Compilot[13] was created and released their beta version for trial by academia and industry. Due to neither Codex nor GitHub Copilot being open-sourced, there are several attempts to reproduce their performance, like PYCODEGPT-CERT [73], CodeParrot[14], and GPT-CC[15]. Encoder-decoder pre-trained models are composed of an encoder and a decoder. AlphaCode [37], which is pre-trained through GitHub repositories with 715.1 GB of code, uses an encoder-decoder transformer architecture. It achieves on average a ranking in the top 54% in competitions with more than 5,000 participants in simulated evaluations.

ChatGPT, a language model developed by the team of OpenAI, has the potential to play a role in code generation. As it is widely known, ChatGPT offers a chat window to enable interaction in a conversational way. In addition to its powerful capabilities for natural language processing tasks, ChatGPT inherits the code generation capabilities from Codex and can perform even better, so the OpenAI team has announced the deprecation of Codex series models in its official documents. There are several research works that mentioned its ability in code-related areas, including mathematical capability [18], bug-solving capability [62], and software testing [29]. ChatGPT's 'Regenerate response' function demonstrates the diversity of its output, but at the same time, it also raises concerns about the consistency of its output given the same input. Currently, people are amazed by its superficial

[12]https://openai.com/blog/openai-codex
[13]https://github.com/features/copilot
[14]https://huggingface.co/codeparrot/codeparrot
[15]https://github.com/CodedotAl/gpt-code-clippy

performance in terms of code generation, however, there is still no research work focused on the threat of non-determinism. Therefore, we think it is necessary to make a comprehensive evaluation of ChatGPT's ability in code generation. More detailed information could be found on its official website's blog [2].

6.3 Non-determinism Handling in the Literature

The non-determinism issue has been studied in traditional Deep Learning-related research: Pham et al. [53] measure the influence of nondeterminism-introducing (NI)-factors in Deep Learning, and study the awareness of this variance among researchers and practitioners. However, the severity of the non-determinism threat in LLM-based coding studies remains unclear.

To understand how well LLM-based code generation papers handle the threat of non-determinism, we collect research articles from Google Scholar with the query 'code generation' AND 'Large Language Model' in the past 2 years (from January 2022 to July 2023). During the search, we search the full text of the paper (excluding citations and appendixes) for keywords, such as non-determinism and its synonyms, the number of experimental repetitions, and the variance of experimental results. After locating these keywords, we manually combine the context to confirm whether the sentence means to declare that non-determinism exists in their study. If the statement exists in the experimental section of the paper and the authors consider non-determinism in their experiment setting and result report, we classify it as considering non-determinism in the experimental design and mentioning non-determinism in the paper; otherwise, if non-determinism is mentioned elsewhere without any actions to mitigate non-determinism, such as in the discussion section, we classify it as only mentioning non-determinism, but not considering this factor in the experiment. If the above keywords are not mentioned in the paper, we read the full text of the paper to ensure that there are no sentences mentioning non-determinism in the paper. If relevant non-determinism statements were encountered, we classify the paper using the above classification method and update our keyword library. After ensuring that our keyword database is up to date and that the two search results are consistent, we searched all the papers twice to obtain our literature review data.

There are 107 papers obtained from Google Scholar according to their relevance rankings. In this survey, we mainly focus on articles with experiments and exclude those with posters and visions only, which yields a set of 76 papers. After an in-depth reading of the experimental design and discussion in each paper, we find that only 35.5% (27/76) out of the 76 papers mention non-determinism or related terms (e.g., stability, randomness, and variance) in their papers. Among them, 21.1% (16/76) papers consider non-determinism in their experimental evaluation, including fixed random seeds, multiple runs of experiments with different fixed random seeds, and report results with error bars or standard deviation. In addition, 14.5% (11/76) of the papers do not consider non-determinism in their experiments, but discuss the threat of non-determinism in their paper.

7 DISCUSSION

In this section, we discuss the implications, trade-offs of non-determinism, and future research directions for code generation with LLMs.

7.1 Implications for Software Developers and Researchers

For developers, it is essential to recognize the limitations of ChatGPT and the potential risks of using generated code in production. If developers prefer a more stable code, they can use a smaller temperature but should keep in mind that even the smallest temperature (i.e., *temperature*=0) could not guarantee the determinism. Moreover, our observation on the correlation between the length of prompts and code correctness/non-determinism suggests the importance of prompt engineering. Developers should thoroughly test the generated code before deploying it, and even consider incorporating more robust testing and validation processes to ensure the determinism and reliability of the generated code.

For researchers, the variance of the generated code raises questions about the quality and validity of the results obtained from assessing LLMs in code generation. If the code generated from ChatGPT is unstable, it can lead to non-reproducible results and unreliable conclusions. Therefore, researchers should carefully consider the limitations of ChatGPT when designing experiments and interpreting results. To reduce the randomness caused by the non-determinism issue, researchers can report the average results, variance, or distribution from multiple requests. Also, it is important to use different datasets, since our study finds that both the correctness and non-determinism of the generated code vary significantly from dataset to dataset. In addition, using a prompt with detailed instructions, a clear structure, and concrete response requirements would help to reduce randomness in generated code.

7.2 Trade-off of non-determinism

Our empirical study highlights the issue of non-determinism in code generation tasks when using ChatGPT. While we underscore the challenges this non-determinism introduces, particularly in terms of ensuring consistency and reliability in generated outputs, it is essential to also acknowledge the potential benefits that non-determinism brings, especially in the realm of creativity.

The inherent non-deterministic nature of LLMs can foster a degree of creativity and diversity in the outputs that deterministic systems may not achieve. This aspect is particularly valuable in applications requiring innovative solutions or creative content generation, where the variety and uniqueness of the output are more critical than in strictly rule-based or deterministic scenarios. In other words, the non-determinism implies that making multiple requests to LLMs may increase the chance for developers to receive high quality code and therefore enhance the code generation performance. For instance, through making five requests in RQ1 with temperature of 1, the candidate that achieves the highest pass rate for a given code problem shows an improvement on average of around 16.13 times (CodeContests), 3.12 times (APPS), and 1.98 times (HumanEval) over the candidate with the lowest pass rate; it exhibits an overall improvement of 5.21 times (CodeContests), 1.40 times (APPS), and 0.59 times (HumanEval) against its mean performance among five candidates. Looking deeper into the consistency of the error, we can find that generated code candidates are more likely (at least 65.85%, 73.83%, and 90.00% in CodeContests, APPS, and HumanEval) to share the same error type if all of them fail to pass the test cases. The most common error types they share are IndexError (46.03% in CodeContests), IndexError (34.78% in APPS), and NameError (33.33% in HumanEval) respectively, under temperature=0.

7.3 Future work

Achieving an optimal balance between determinism and creativity is crucial for enhancing LLMs' effectiveness across a broad spectrum of applications. Too much determinism could stifle creativity, leading to predictable and monotonous outputs, while excessive non-determinism might compromise the reliability and consistency necessary for applications requiring precise and accurate results. To address these challenges and strike a balance between determinism and creativity, future research could explore several promising directions:

Voting Mechanism: Implementing a voting mechanism wherein multiple candidates of the model generate outputs, and a consensus approach should be used to select the most appropriate output. This method can help mitigate the effects of non-determinism by leveraging the collective decision-making process to choose outputs that are both creative and relevant to the task.

Repair Loop Driven by LLMs: Developing techniques for loop repair driven by LLMs can offer a novel approach to addressing non-determinism. By automatically identifying and correcting inconsistencies or errors in the generated code, such a system could enhance the reliability of outputs without significantly compromising creativity. This approach would rely on the model's ability to learn from feedback loops, improving its performance over time.

Hybrid Models: Investigating hybrid models that combine deterministic and non-deterministic components might offer a pathway to achieving the desired balance. Such models could leverage the strengths of both approaches, using deterministic methods to ensure reliability and consistency where needed, while allowing for creative freedom through non-deterministic processes in aspects where innovation is prized.

Customizable Levels of Determinism: Developing LLMs that allow users to specify their preferred level of determinism versus creativity could cater to a wide range of applications. This customization could enable users to tune the model's outputs according to the specific requirements (e.g. domain-specific) of their task, whether that be generating highly creative content or producing consistent and reliable code.

8 CONCLUSION

This work studies the non-determinism threat of code generation with ChatGPT. We perform experiments on three widely studied code generation benchmarks and find that the correctness, test outputs, as well as syntax and structure of code candidates generated from the same instruction, vary significantly in different requests. We hope that this paper could raise awareness of the threat of non-determinism in future code generation tasks when using large language models.

9 ACKNOWLEDGEMENT

This work was supported by the UKRI Centre for Doctoral Training in Safe and Trusted Artificial Intelligence (EP/S023356/1).

REFERENCES

[1] https://152334h.github.io/blog/non-determinism-in-gpt-4/.
[2] https://chat.openai.com/chat.
[3] https://github.com/ShuyinOuyang/LLM-is-a-box-of-chocolate.
[4] 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).
[5] 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).
[6] Y Bang, S Cahyawijaya, N Lee, W Dai, D Su, B Wilie, H Lovenia, Z Ji, T Yu, W Chung, et al. 2023. A multitask, multilingual, multimodal evaluation of ChatGPT on reasoning, hallucination, and interactivity. arXiv.
[7] Bhavya Bhavya, Jinjun Xiong, and Chengxiang Zhai. 2022. Analogy generation by prompting large language models: A case study of instructgpt. *arXiv preprint arXiv:2210.04186* (2022).
[8] Alan W Biermann. 1978. The inference of regular LISP programs from examples. *IEEE transactions on Systems, Man, and Cybernetics* 8, 8 (1978), 585–600.
[9] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.
[10] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, et al. 2023. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712* (2023).
[11] Subhashis Chatterjee, Deepjyoti Saha, Akhilesh Sharma, and Yogesh Verma. 2022. Reliability and optimal release time analysis for multi up-gradation software with imperfect debugging and varied testing coverage under the effect of random field environments. *Annals of Operations Research* (2022), 1–21.
[12] 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.(2021). *arXiv preprint arXiv:2107.03374* (2021).
[13] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2023. Large language models are edge-case fuzzers: Testing deep learning libraries via fuzzgpt. *arXiv preprint arXiv:2304.02014* (2023).
[14] Li Dong and Mirella Lapata. 2016. Language to logical form with neural attention. *arXiv preprint arXiv:1601.01280* (2016).
[15] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. 2023. Large Language Models for Software Engineering: Survey and Open Problems. *arXiv preprint arXiv:2310.03533* (2023).

[16] Yunhe Feng, Sreecharan Vanam, Manasa Cherukupally, Weijian Zheng, Meikang Qiu, and Haihua Chen. 2023. Investigating code generation performance of ChatGPT with crowdsourcing social data. In *2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 876–885.
[17] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).
[18] Simon Frieder, Luca Pinchetti, Ryan-Rhys Griffiths, Tommaso Salvatori, Thomas Lukasiewicz, Philipp Christian Petersen, Alexis Chevalier, and Julius Berner. 2023. Mathematical capabilities of chatgpt. *arXiv preprint arXiv:2301.13867* (2023).
[19] Cordell Green. 1981. Application of theorem proving to problem solving. In *Readings in Artificial Intelligence*. Elsevier, 202–222.
[20] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices* 46, 1 (2011), 317–330.
[21] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. 2017. Program synthesis. *Foundations and Trends® in Programming Languages* 4, 1-2 (2017), 1–119.
[22] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366* (2020).
[23] Qi Guo, Junming Cao, Xiaofei Xie, Shangqing Liu, Xiaohong Li, Bihuan Chen, and Xin Peng. 2024. Exploring the potential of chatgpt in automated code refinement: An empirical study. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.
[24] Tatsunori B Hashimoto, Kelvin Guu, Yonatan Oren, and Percy S Liang. 2018. A retrieve-and-edit framework for predicting structured outputs. *Advances in Neural Information Processing Systems* 31 (2018).
[25] Hossein Hassani and Emmanuel Sirmal Silva. 2023. The role of ChatGPT in data science: how ai-assisted conversational interfaces are revolutionizing the field. *Big data and cognitive computing* 7, 2 (2023), 62.
[26] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. 2021. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938* (2021).
[27] Dan Hendrycks, Collin Burns, Steven Basart, Andrew Critch, Jerry Li, Dawn Song, and Jacob Steinhardt. 2020. Aligning ai with shared human values. *arXiv preprint arXiv:2008.02275* (2020).
[28] Jeevana Priya Inala, Chenglong Wang, Mei Yang, Andres Codas, Mark Encarnación, Shuvendu Lahiri, Madanlal Musuvathi, and Jianfeng Gao. 2022. Fault-aware neural code rankers. *Advances in Neural Information Processing Systems* 35 (2022), 13419–13432.
[29] Sajed Jalil, Suzzana Rafi, Thomas D LaToza, Kevin Moran, and Wing Lam. 2023. Chatgpt and software testing education: Promises & perils. In *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 4130–4137.
[30] Andrej Kiviriga. 2023. Efficient Model Checking: The Power of Randomness. (2023).
[31] Kalpesh Krishna, Yapei Chang, John Wieting, and Mohit Iyyer. 2022. Rankgen: Improving text generation with large ranking models. *arXiv preprint arXiv:2205.09726* (2022).
[32] Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. 2019. Spoc: Search-based pseudocode to code. *Advances in Neural Information Processing Systems* 32 (2019).
[33] Emanuele La Malfa, Aleksandar Petrov, Simon Frieder, Christoph Weinhuber, Ryan Burnell, Anthony G Cohn, Nigel Shadbolt, and Michael Wooldridge. 2023. The ARRT of Language-Models-as-a-Service: Overview of a New Paradigm and its Challenges. *arXiv preprint arXiv:2309.16573* (2023).
[34] Mina Lee, Percy Liang, and Qian Yang. 2022. Coauthor: Designing a human-ai collaborative writing dataset for exploring language model capabilities. In *Proceedings of the 2022 CHI conference on human factors in computing systems*. 1–19.
[35] Jia Li, Ge Li, Zhuo Li, Zhi Jin, Xing Hu, Kechi Zhang, and Zhiyi Fu. 2023. Codeeditor: Learning to edit source code with pre-trained models. *ACM Transactions on Software Engineering and Methodology* 32, 6 (2023), 1–22.
[36] Jia Li, Yongmin Li, Ge Li, Zhi Jin, Yiyang Hao, and Xing Hu. 2023. Skcoder: A sketch-based approach for automatic code generation. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2124–2135.
[37] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science* 378, 6624 (2022), 1092–1097.
[38] Zongjie Li, Chaozheng Wang, Zhibo Liu, Haoxuan Wang, Dong Chen, Shuai Wang, and Cuiyun Gao. 2023. Cctest: Testing and repairing code completion systems. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1238–1250.
[39] Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. 2023. Code as policies: Language model programs for embodied control. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 9493–9500.
[40] Wang Ling, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiskỳ, Andrew Senior, Fumin Wang, and Phil Blunsom. 2016. Latent predictor networks for code generation. *arXiv preprint arXiv:1603.06744* (2016).
[41] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *arXiv preprint arXiv:2305.01210* (2023).

[42] 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).
[43] Yiheng Liu, Tianle Han, Siyuan Ma, Jiayue Zhang, Yuanyuan Yang, Jiaming Tian, Hao He, Antong Li, Mengshen He, Zhengliang Liu, et al. 2023. Summary of chatgpt/gpt-4 research and perspective towards the future of large language models. *arXiv preprint arXiv:2304.01852* (2023).
[44] 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).
[45] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664* (2021).
[46] Zohar Manna and Richard J Waldinger. 1971. Toward automatic program synthesis. *Commun. ACM* 14, 3 (1971), 151–165.
[47] 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. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2149–2160.
[48] Patrick E McKight and Julius Najab. 2010. Kruskal-wallis test. *The corsini encyclopedia of psychology* (2010), 1–1.
[49] Patrick E McKnight and Julius Najab. 2010. Mann-Whitney U Test. *The Corsini encyclopedia of psychology* (2010), 1–1.
[50] Eric Mitchell, Yoonho Lee, Alexander Khazatsky, Christopher D Manning, and Chelsea Finn. 2023. Detectgpt: Zero-shot machine-generated text detection using probability curvature. *arXiv preprint arXiv:2301.11305* (2023).
[51] Prabhat Nagarajan, Garrett Warnell, and Peter Stone. 2018. Deterministic implementations for reproducibility in deep reinforcement learning. *arXiv preprint arXiv:1809.05676* (2018).
[52] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]
[53] Hung Viet Pham, Shangshu Qian, Jiannan Wang, Thibaud Lutellier, Jonathan Rosenthal, Lin Tan, Yaoliang Yu, and Nachiappan Nagappan. 2020. Problems and opportunities in training deep learning software systems: An analysis of variance. In *Proceedings of the 35th IEEE/ACM international conference on automated software engineering*. 771–783.
[54] Gabriel Poesia, Oleksandr Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2022. Synchromesh: Reliable code generation from pre-trained language models. *arXiv preprint arXiv:2201.11227* (2022).
[55] Joan Puigcerver, Carlos Riquelme, Basil Mustafa, and Neil Houlsby. 2023. From Sparse to Soft Mixtures of Experts. *arXiv preprint arXiv:2308.00951* (2023).
[56] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. Improving language understanding by generative pre-training. (2018).
[57] David E Shaw, William R Swartout, and C Cordell Green. 1975. Inferring LISP Programs From Examples.. In *IJCAI*, Vol. 75. 260–267.
[58] David Canfield Smith. 1975. *Pygmalion: a creative programming environment.* Stanford University.
[59] Ioana Baldini Soares, Dennis Wei, Karthikeyan Natesan Ramamurthy, Moninder Singh, and Mikhail Yurochkin. 2022. Your fairness may vary: pretrained language model fairness in toxic text classification. In *Annual Meeting of the Association for Computational Linguistics*.
[60] Phillip D Summers. 1977. A methodology for LISP program construction from examples. *Journal of the ACM (JACM)* 24, 1 (1977), 161–175.
[61] Zeyu Sun, Qihao Zhu, Lili Mou, Yingfei Xiong, Ge Li, and Lu Zhang. 2019. A grammar-based structural cnn decoder for code generation. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 33. 7055–7062.
[62] Nigar M Shafiq Surameery and Mohammed Y Shakor. 2023. Use chat gpt to solve programming bugs. *International Journal of Information Technology & Computer Engineering (IJITC) ISSN: 2455-5290* 3, 01 (2023), 17–22.
[63] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1433–1443.
[64] 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.
[65] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering* (2024).
[66] Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. 2019. Code generation as a dual task of code summarization. *Advances in neural information processing systems* 32 (2019).
[67] Xiongfei Wu, Liangyu Qin, Bing Yu, Xiaofei Xie, Lei Ma, Yinxing Xue, Yang Liu, and Jianjun Zhao. 2020. How are deep learning models similar? an empirical study on clone analysis of deep learning software. In *Proceedings of the 28th International Conference on Program Comprehension*. 172–183.
[68] Frank F Xu, Bogdan Vasilescu, and Graham Neubig. 2022. In-ide code generation from natural language: Promise and challenges. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 2 (2022), 1–47.

[69] Burak Yetiştiren, Işık Özsoy, Miray Ayerdem, and Eray Tüzün. 2023. Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT. *arXiv preprint arXiv:2304.10778* (2023).
[70] Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. *arXiv preprint arXiv:1704.01696* (2017).
[71] Pengcheng Yin and Graham Neubig. 2018. Tranx: A transition-based neural abstract syntax parser for semantic parsing and code generation. *arXiv preprint arXiv:1810.02720* (2018).
[72] Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. 2024. Codereval: A benchmark of pragmatic code generation with generative pre-trained models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–12.
[73] Daoguang Zan, Bei Chen, Dejian Yang, Zeqi Lin, Minsu Kim, Bei Guan, Yongji Wang, Weizhu Chen, and Jian-Guang Lou. 2022. CERT: Continual Pre-training on Sketches for Library-oriented Code Generation. *arXiv preprint arXiv:2206.06888* (2022).