Revisiting the Non-Determinism of Code Generation by the GPT-3.5 Large Language Model

Salimata SAWADOGO
Université Joseph Ki-Zerbo
Centre d'Excellence en IA (CITADEL)
Ouagadougou, Burkina Faso
sdgsalma@gmail.com

Aminata SABANE
Université Joseph Ki-Zerbo
Centre d'Excellence en IA (CITADEL)
Ouagadougou, Burkina Faso
aminata.sabane@ujkz.bf

Rodrique KAFANDO
Université Virtuelle du Burkina Faso
Centre d'Excellence en IA (CITADEL)
Ouagadougou, Burkina Faso
rodrique.kafando@citadel.bf

Abdoul Kader KABORE
Université du Luxembourg
& Centre d'Excellence en IA (CITADEL)
Ouagadougou, Burkina Faso
abdoulkader.kabore@uni.lu

Tegawendé F. BISSYANDE
Université du Luxembourg
& Université Joseph Ki-Zerbo
Luxembourg, Luxembourg
tegawende.bissyande@uni.lu

***Abstract*—Despite recent advancements in Large Language Models (LLMs) for code generation, their inherent non-determinism remains a significant obstacle for reliable and reproducible software engineering research. Prior work has highlighted the high degree of variability in LLM-generated code, even when prompted with identical inputs. This non-deterministic behavior can undermine the validity of scientific conclusions drawn from LLM-based experiments. This paper showcases the Tree of Thoughts (ToT) prompting strategy as a promising alternative for improving the predictability and quality of code generation results. By guiding the LLM through a structured Thoughts process, ToT aims to reduce the randomness inherent in the generation process and improve the consistency of the output. Our experiments on GPT-3.5 Turbo model using 829 code generation problems from benchmarks such as CodeContests, APPS (Automated Programming Progress Standard) and HumanEval demonstrate a substantial reduction in non-determinism compared to previous findings. Specifically, we observed a significant decrease in the number of coding tasks that produced inconsistent outputs across multiple requests. Nevertheless, we show that the reduction in semantic variability was less pronounced for HumanEval (69%), indicating unique challenges present in this dataset that are not fully mitigated by ToT.**

***Index Terms*—Code Generation, Tree of Thoughts, LLMs, Non-Determinism**

I. Introduction

In the current era of artificial intelligence (*AI*), large language models (*LLM*) have revolutionized the field of natural language processing. One particularly impacted domain is code generation, where *LLM* offers unprecedented capabilities to automate and facilitate computer programming. Software development increasingly relies on the promising capabilities of large language models (LLM) to automate and simplify programming [1]. These models, capable of generating code from natural language queries, offer immense potential to enhance developer productivity and efficiency [2]. However, this technological advancement comes with challenges, particularly their tendency to produce non-deterministic results [3]. Indeed, LLMs are designed to explore multiple potential solutions for a given query, which can lead to variable and unpredictable outcomes [4]. In the context of code generation, this uncertainty poses a significant issue, as accuracy and reliability are critical [5]. Non-deterministically generated code may contain errors, inconsistencies, or inefficiencies, limiting its practical utility [6].

This study focuses on reproducibility and aims to evaluate the potential of the *Tree of Thoughts* (ToT) approach to mitigate the non-determinism challenges hindering the adoption of LLMs in critical production environments. Introduced by Yao et al. (2023) [7], the ToT approach draws inspiration from human reasoning mechanisms by structuring queries to LLMs, thereby fostering more predictable and coherent responses. This technique generalizes the *Chain-of-Thoughts* (CoT) method [8] by encouraging LLMs to explore coherent sequences of intermediate steps, termed "thoughts," in their problem-solving process. With this approach, LLMs can anticipate or revise their decisions, enabling more robust and deterministic solutions.

In this context, this paper thoroughly explores the application and evaluation of the ToT approach in the field of code generation, emphasizing its reproducibility and its ability to enhance the reliability of LLMs. This research aims not only to demonstrate the effectiveness of this approach but also to identify limitations and opportunities for future work in this domain.

The structure of this paper is organized as follows: Section 2 presents a comprehensive analysis of the existing literature, highlighting current approaches, their limitations, and research opportunities. Based on this analysis, Section 3 proposes an innovative approach to address the identified challenges. Concrete experiments, described in Section 4, evaluate the effectiveness of our approach and compare it to existing methods. Results are analyzed and discussed in Section 5, and a general conclusion is provided in Section 6.

II. Related work

A. Code generation

The success of *LLMs* in natural language modeling has sparked significant interest among researchers in leveraging these models for code generation [9]. Existing research on code generation models can be categorized into three main approaches: sequence-based techniques, abstract syntax tree (AST)-based methods, and pre-trained models. ***Sequence-based techniques*** view code as a sequence of tokens and use LLM to generate source code, one token at a time, based on input descriptions. Examples include generative models with character-level softmax and a multi-pointer network proposed by Ling and al. [10] and retrieval models with a noisy encoder-decoder [11].

Tree-based method, which generates a code analysis tree, such as an abstract syntax tree (AST), from input descriptions. Notable examples include the work of Dong *et al.*, who encode statements into vectors and generate their corresponding logical forms as trees using Long Short-Term Memory (LSTM) models, as well as the semantic parser "Tranx" by Yin *et al.*, which produces the sequence of tree construction actions through a transition-based neural model and builds the AST from this sequence of actions [12]. ***Pre-trained models***, trained on extensive amounts of source code, can be fine-tuned on specific datasets for code generation tasks. Encoder models, such as CodeBERT [13], are trained with dual objectives: masked language modeling and token replacement detection. Decoder models, such as GPT models [14], predict the next token based on a given input context. Examples include CodeGPT, developed by Lu and al; [15]. for code completion and text-to-code generation, and CodeX12, released in beta for testing by academia and industry.

B. (Non) determinism

To gain insights into how research papers on LLM-based code generation handle the non-determinism threat, Ouyang and al. (2023) [3] conducted a thorough search of 107 papers on LLM-based code generation, focusing on empirical experiments (76/100). They found that only 35.5% of the papers explicitly mentioned non-determinism or related terms, while only 21.1% (16/76) incorporated non-determinism considerations into their experimental evaluations. This highlights the need for increased awareness and effective mitigation strategies regarding non-determinism in LLM-based code generation. At the heart of our choice of the Tree of Thoughts (ToT) approach for code generation lies its unique ability to equip large language models (LLM) with a deep understanding of code context and structure. This deep understanding proves to be the cornerstone of accurate and coherent code generation, thus meeting the crucial requirements of this rapidly evolving field. The majority of existing papers fail to acknowledge or address non-determinism in their experimental evaluations, potentially jeopardizing the reproducibility and reliability of their findings. As the field of LLM-based code generation continues to evolve, it is crucial for researchers to prioritize a deeper understanding of non-determinism and develop robust techniques to minimize its impact. By embracing these measures, the scientific community can foster a more rigorous and trustworthy landscape for LLM-based code generation research.

C. Tree of Thoughts

The *Tree-of-Thoughts (ToT)* prompting is an innovative technique that samples continuous language sequences for problem-solving [7]. *ToT* actively maintains a tree of Thoughts, where each Thoughts is a coherent language sequence that serves as an intermediate step toward problem resolution [16]. By adopting this approach, it allows language models to demonstrate advanced reasoning capabilities. *ToT* enables LLMs to autonomously correct their mistakes and continuously accumulate knowledge, resulting in improved performance and better decision-making.

The Tree of Thoughts (ToT) approach offers a novel solution that draws inspiration from human cognitive processes. By simultaneously taking into account multiple potential courses of action, ToT enables LLM to assess the implications of each decision and optimize their choices [17]. This method fits into a larger research trend aimed at enhancing the quality of responses produced by LLM, such as by incorporating mechanisms for self-evaluation and criticism [18]. The concept of injecting model-generated returns to refine results has already been explored in previous works [19], or the LLM can be integrated into algorithmic search processes where search trees are expanded with relevant paragraphs that may offer answers. ToT, however, stands out for its ability to handle challenging tasks and search a larger space for solutions. In fact, ToT enables LLM to simulate several paths and identify the most promising ones by relying on decision trees. As a result, our proposal fits into the overall framework of these studies while also making unique additions, such as in terms of flexibility and the ability to handle non-deterministic problems.

In this literature review, we addressed the work of Ouyang and al. [3] which highlighted the importance of controlling non-determinism, optimizing prompts, and rigorously testing generated code to mitigate risks and improve software engineering outcomes. The "Zero-Shot" prompting has been used to reduce this variability, but its limits have been acknowledged, in terms of exploiting the intrinsic capabilities of LLM. A more sophisticated alternative, the Tree of Thoughts (ToT), has emerged as a promising method to improve the predictability and quality of code generation results. This paper aims to contribute to the scientific literature by conducting rigorous experiments to evaluate the determinism of LLM in code generation using the ToT method.

III. METHOD

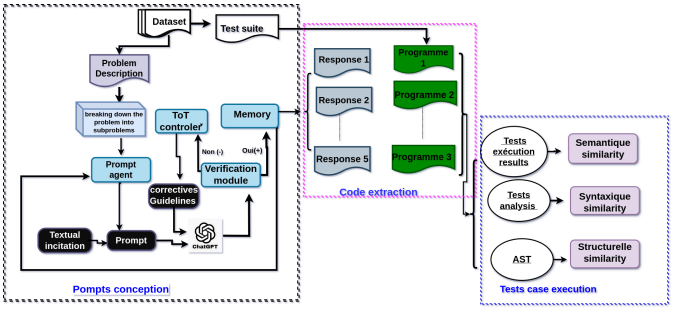

Fig. 1. Overview of the experimental approach.

Our approach draws inspiration from prior research that has highlighted the stochastic nature of code generated by large language models (LLMs). In contrast to these studies, which primarily sample continuous language sequences for problem-solving, our Tree of Thought (ToT) approach actively maintains a structured tree of thoughts, where each thought represents a coherent language sequence serving as an intermediate step toward problem resolution [16]. This strategy enables structured and guided reasoning at every stage of generation. Our results demonstrate that this method significantly reduces variability among the codes generated from the same prompt, representing a substantial improvement over the approach proposed by Ouyang and al. [20]. A comprehensive summary of our experimental procedure is presented in Fig. 2. For each code generation problem, we design a prompt accompanied by a coding instruction, which is subsequently submitted to the ChatGPT API as part of the Tree of Thought (ToT) prompting approach. To enable ChatGPT to produce five distinct predictions from the same prompt, we perform five successive API calls. The codes extracted from these five responses, referred to as candidate codes, are then subjected to a thorough analysis of their syntactic, structural, and semantic similarities. The evaluation of a method leveraging ToT prompting to enhance determinism in code generation requires a rigorous experimental framework. This methodology is grounded in the careful selection of benchmarks, precise model adaptation, and the use of relevant performance metrics.

Design of ToT Prompting: In the initial phase of our study, we pay particular attention to the design of prompts using the *ToT* method. Unlike the standard methods of evaluating code generation by *LLMs* [20], where a simple *zero-shot* prompt is used, our *ToT* approach involves creating structured prompts in multiple steps to guide the *LLM* in generating correct and deterministic code. For each programming problem, our prompt begins with an analysis phase, where we ask targeted questions to encourage *ChatGPT* to think about the appropriate algorithmic or logical approach. Next, we move to a planning phase, asking the model to outline the key steps or logic necessary for Python implementation. Finally, we transition to the generation phase, where we formulate an explicit and structured request for generating Python code in 'Markdown' format, to facilitate the extraction and distinction of code from natural language in the *API* response. Thus, our *ToT* prompt will look like this:
Step 1: Analyze the task (problem).
Step 2: Plan the coding steps in Python (breaking down the problem into sub-problems).
Step 3: Generate Python 3 code in Markdown format. This multi-step structure allows for more targeted and structured code generation aligned with the objectives of our research.

Code Extraction: After receiving the response from *ChatGPT*, we proceed with extracting the code from the generated text. The code is then compiled in its original form, without making any modifications.

Test Case Execution To assess the semantics of the code produced by the *LLM*, we use a test suite tailored for each benchmark. We record not only whether each test succeeded or failed, but also each individual test result. This evaluation allows us to examine the similarity of the test results even if they are all zero. For the *CodeContests* and *HumanEval* datasets, each problem has a time threshold of nine (9) seconds. The entire dataset in *APPS* does not automatically provide a time value; instead, we set it to nine (9) seconds. To ensure that test cases are executed sequentially and to prevent concurrency issues that could arise from simultaneous executions, we use scripts called *monothread* (instructions are executed in the order they are written). We introduce similarity measurement tools to assess the semantic, syntactic, and structural similarity between the generated code solutions in order to gauge the degree of similarity between the candidate codes. We used the *GPT-3.5 Turbo* model for our experiments to evaluate semantic, syntactic, and structural similarity between code candidates. Semantic similarity is measured by comparing test execution results, while syntactic similarity is measured by comparing textual similarity between codes. Structural similarity is evaluated by comparing the abstract syntax trees (AST) of the code candidates. We have identified three research issues from this experimental study, which we will discuss in the next section.

A. Use cas: Resolving the Fibonacci Problem Using the Tree of Thought (ToT) Method

1) Problem Introduction: Clear Formulation: The problem is articulated clearly and succinctly using natural language that the LLM can process. For instance: "What is the value of the 10th term in the Fibonacci sequence?" Input to the LLM: The problem formulation is introduced into the LLM in textual form.
2) Problem Decomposition: dentifying Sub-problems: The LLM identifies inherent sub-problems within the Fibonacci sequence, including:the definition of the sequence, the recurrence relation, calculating preceding terms. Node Creation:Each sub-problem is represented as a node within the Tree of Thought.

3) Solution Exploration: Hypothesis Generation: For each node, the LLM generates multiple hypotheses or potential solutions. For example: To calculate the 10th term, it could consider explicit formula usage, recurrence relations, or iterative loops. Hypothesis Evaluation: The LLM assesses each hypothesis based on: Relevance, Consistency with other hypotheses. Likelihood of leading to the correct solution.
4) Tree Construction: Branch Creation:Different hypotheses and their evaluations are linked together, forming the branches of the Tree of Thought. Hierarchical Structure: The tree develops hierarchically, with the initial problem at the top and detailed solutions branching out below.
5) Selecting the Best Solution: lobal Evaluation: The LLM evaluates the entire tree to identify the most promising branch—one that leads to the most probable and efficient solution. Final Solution Selection: The LLM follows the chosen branch to select the final solution.
6) Generating the Response Response Formulation: The LLM formulates a clear and concise response using natural language, presenting the final solution to the Fibonacci problem.

The figure is an illustration with the Fibonacci sequence:

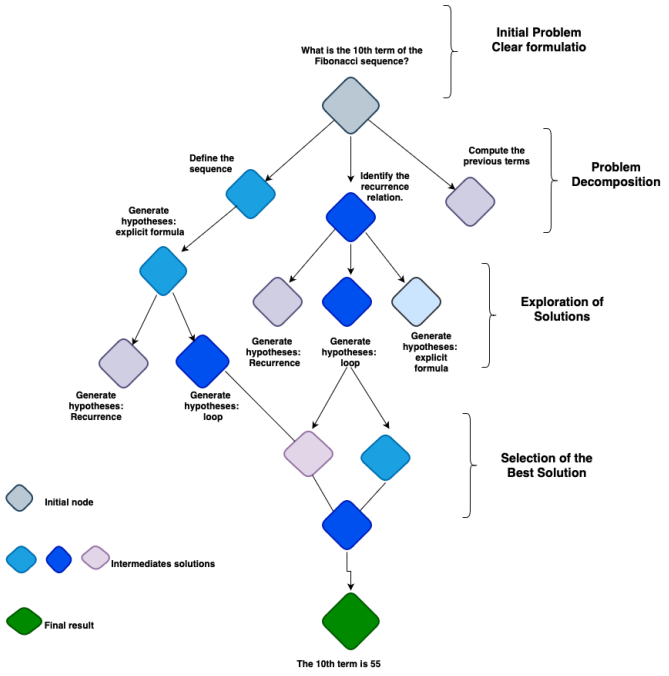

Fig. 2. Overview of an illustration with the Fibonacci sequence

IV. Experimental Design

A. Research Questions

This study addresses the following research questions:

- *RQ1: To what extent does the use of ToT impact ChatGPT ability to handle non-determinism in code generation?* This RQ investigates the non-determinism of ChatGPT in terms of semantic, syntactic, and structural similarity between the candidate codes generated with identical prompts. There are three subquestions:
  - Sub-RQ1.1: To what extent is GPT sensitive to non-determinism in terms of semantic similarity?
  - Sub-RQ1.2: To what extent is ChatGPT sensitive to non-determinism in terms of syntactic similarity?
  - Sub-RQ1.3: To what extent is ChatGPT sensitive to non-determinism in terms of structural similarity?
- RQ2: *How does temperature affect the degree of non-determinism when generating code by ToT? Temperature is a hyperparame*ter of LLM that allows to control the randomness of predictions. This RQ checks and compares the non-determinism of ChatGPT in code generation with different temperature choices.
- RQ3: *How does non-determinism compare to the similarity of the best candidate codes generated in the same prediction?*

ChatGPT can be configured to generate multiple candidate codes for a prediction, ranked by their predictive probability. This RQ compares the similarity of the candidate codes obtained in different predictions with those obtained within the same prediction.

B. Code Generation Benchmark

Our experiments use the three most studied code generation benchmarks. :*CodeContest* [21], *APPS* [22] and *HumanEval* [8]. Table I shows the details of each reference. We also provide further details below:

TABLE I
Code Generation Benchmark

Name	Average Description Length	Number of Problems	Average Number of Test Cases	Average Number of Correct Codes
CodeContests [21]	1989.19	80.43	203.84	49.99
APPS [22]	1663.94	500	80.43	20.92
HumanEval [8]	450.60	164	9.24	1.00

CodeContest:CodeContests[1] is a competitive programming dataset for machine learning. It was used in the training of AlphaCode and includes programming problems from various sources such as Aizu, AtCoder, CodeChef, Codeforces, and HackerEarth. The problems include test cases in the form of paired inputs and outputs, as well as correct and incorrect human-written solutions in a variety of languages. In our experiments following the evaluation practice of AlphaCode, we use the CodeContests test set for the code generation tasks of our LLM.

APPS: The APPS dataset [2] consists of problems collected from various freely accessible coding websites such as Codeforces, Kattis, etc. The APPS benchmark aims to reflect how human programmers are assessed by posing coding problems in natural language without restrictions and evaluating the accuracy of the solutions. The problems measure coding ability

[1]https://paperswithcode.com/dataset/codecontests
[2]https://paperswithcode.com/dataset/apps

as well as problem-solving skills. It includes 10,000 coding problems with 131,836 test cases to verify solutions and 232,444 human-written ground truth solutions. The data is evenly split into training and test sets, each containing 5,000 problems. In the test set, each problem has multiple test scenarios, with an average of 80.43 test scenarios per problem. Each test case is specifically designed for the corresponding problem, allowing us to rigorously assess program functionality. This dataset is exclusively designed for evaluating Python program synthesis. The original test set contains 5,000 problems, and we randomly sample 500 problems.

HumanEval: The HumanEval dataset [3] is an evaluation dataset for the HumanEval problem-solving dataset described in the paper [8]. It was used to measure functional accuracy of program synthesis from docstrings. It consists of 164 original programming problems evaluating language understanding, algorithms, and simple mathematics, some of which are comparable to simple software interview questions. Each problem includes a function signature, a docstring, a body, and several unit tests with an average of 9.24 test cases per problem. We use this dataset to compare our experiments.

C. Performance Metrics

In order to answer our research questions, we introduce the following tools to measure the degree of non-determinism:

Semantic Similarity: The semantic similarity of candidate codes is measured using test execution results that include test pass rate and output equivalence rate (OER). The test pass rate is a widely used metric to evaluate code generation capabilities, with five test pass rates for each candidate code. The OER records the ratio between equivalent test outputs and total test outputs. The paper uses an OER to represent the output equivalence rate and an OER (no ex.) to represent the output equivalence rate without exceptions or errors. Each coding problem has five test pass rates represent in Table II, and the variance and maximum difference of these values are used to check the non-determinism of ChatGPT in code generation. The average value represents the average correctness of the generated code, while the "Worst Case Ratio" shows the ratio of problems with a diff max test pass rate of 1 or an OER of 0.

Syntactic Similarity: The textual similarity of candidate codes is evaluated using the longest common subsequence (LCS) and the Levenshtein edit distance (LED) as evaluation tools. LCS measures similarity by normalizing the length of the longest common subsequence between two sequences, while LED measures the minimum number of token modifications needed to transform one code into another. Both LCS and LED consider tokens as the smallest unit, and measure syntactic similarity between candidate codes. Each code generation problem has four values for each metric, with the average and worst values indicating syntactic similarity in TableIII. Here are the formulas for LCS and LED [3]:

$$LCS = \frac{len(lcs(s,t))}{len(s)} \tag{1}$$

[3]https://paperswithcode.com/dataset/humaneval

where "s" is the reference string, "t" is the string to compare, and "lcs(s, t)" is the longest common subsequence between "s" and "t".

$$LEDs_{t(i,j)} = \begin{cases} \max(i,j) if \min(i,j) = 0 \\ LED_{s,t}(i-1,j) + 1 else \\ LED_{s,t}(i,j-1) + 1 else \\ LED_{s,t}(i-1,j-1) + 1_{(\#t_j)} \end{cases}$$

[3]

where LED_s, t (i, j) represents the Levenshtein edit distance between the first "i" characters of "s" and the first "j" characters of "t", and diff_(si , tj) equals "0" if the i-th character of "s" is identical to the j-th character of "t", and 1 otherwise.

Structural Similarity: Structural similarity measures assess code similarity based on their Abstract Syntax Tree (AST). In our experiment, structural similarity is primarily measured using the $pycode_similar$ tool with two different configurations, namely $United_Diff$ and $Tree_Diff$ (more details in Section 1.3). For the five candidate codes for each coding problem, we use the first candidate code as a reference and calculate the structural similarity between the first candidate code and the four other candidates under the $United_Diff$ and $Tree_Diff$ configurations. Thus, each problem has four values for $United_Diff$ and $Tree_Diff$ respectively, each value indicating a structural similarity metric. We use the average of these four values as well as the worst among them (i.e., the smallest value for$United_Diff$ and $Tree_Diff$) to represent the structural similarity of each problem. The table shows the average and worst values under United $United_Diff$ and $Tree_Diff$ configurations for all coding problems in a dataset.

V. CHATGPT

Among the different language models we studied, we chose *ChatGPT*, which has gained significant popularity and recognition in various tasks, including answering questions, machine translation, sentiment analysis, code generation, and text summarization. Code generation is one of the most impressive tasks. There are several reasons why we selected *ChatGPT* as the target of our research among all these large language models.

First, *ChatGPT* has the ability to generate highly coherent and contextually appropriate words and responses to a wide variety of text prompts. This makes it an ideal tool for conducting research in code generation by designing specific prompts.

Secondly, the *GPT-3.5* series is particularly attractive due to its impressive performance and large-scale training possibilities, which result in more precise and nuanced language processing capabilities.

Thirdly, the *gpt-3.5-turbo* and *gpt-4* API models that were released have not been extensively studied, and their code generation capabilities are not well known. That is why we chose them as targets for our experiment.

As stated on the official *ChatGPT* website, using the *ChatGPT API* requires several parameters. We use the default values for most of the parameters, along with the following:

- **model**: The ID of the model to use, such as *gpt-3.5-turbo* and *text-davinci-003*. This parameter is strictly necessary, and in our case, we assign it the value *gpt-3.5-turbo*.
- **message**: A list of messages describing the conversation so far, where two key values, 'role' and 'content', must be provided. This parameter is also strictly necessary. In our experiments, the "role" message is "user", and the "content" contains the prompt used for the request.
- **temperature**: The sampling temperature to use, between 0 and 2 (the 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 examine the influence of temperature in RQ3 with three temperature values: 0, 1, and 2. For other RQ, we use the default temperature value (i.e., temperature=1).
- **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 this into account and set this value to its default (i.e., top.p=1).
- **n**: The number of chat completion choices to generate for each input message (the default value is 1). This parameter is specifically used for question 3, which compares the differences between various ways of requesting information. We chose n=5, as five is a commonly used number in variance studies.

We generate five responses separately by asking *ChatGPT* five times to generate Python code, but we only take the first response from each request as a reference.

VI. Findings & Discussion

A. RQ1.1 Semantic Similarity

This section presents the experimental results along with the analysis and discussion for each research question (RQ). For our evaluation, we measure semantic similarity using the following metrics: the test pass rate, the Output Equivalence Rate (OER), as well as the OER excluding exceptions (OER no.ex). As mentioned in our method, each coding problem has five test pass rates. We use the variance and the maximum difference of these five values to verify the Non-determinism of ChatGPT in code generation for the task. We also report the average value, which represents the average correctness of the generated code. For the OER or OER (no. ex), each coding problem has a single value, indicating the similarity of the test outputs across the five code candidates. The average values of the measurements for all coding problems in a dataset are presented in Table II. The *max diff* refers to the maximum difference value among all the coding problems. Additionally, Table II also shows the "Worst-case Ratio," which is the ratio of problems with a max diff in test pass rate.

TABLE II
RQ1.1 Semantic Similarity Results (SS). OER and OER (no ex.) are respectively the Output Equivalence Rate and the Output Equivalence Rate excluding exceptions.

Semantic similarity	Metrics	CodeContests	APPS	HumanEval
Test pass rate	Mean	0.14	0.18	0.80
	Mean Variance	1.57	6.40	7.60
	Diff max.	0.00	0.00	0.00
	Ratio of worst cases	4.32%	9%	69%
OER	Mean	1.00	1.00	0.99
	Ratio of worst cases	0%	0%	0%
OER (no ex.)	Mean	0.78	0.49	0.94
	Ratio of worst cases	13%	42.2%	3%

Table II shows percentages of coding tasks without a single identical test output among the five different requests of 4.32%; 9%; and 69% for the CodeContests, APPS, and HumanEval datasets, respectively. These results show a drastic reduction in semantic variability for the CodeContests and APPS datasets compared to the results of Ouyang and al. [3] which show percentages of 72.73%; 60.40% for the same datasets.

This improvement indicates significantly increased consistency in code generation, showing that TOT enhances code generation consistency by providing better understanding of prompt instructions and resulting in predictable outputs, enhancing the fit between provided instructions and generated code. For HumanEval, the reduction in semantic variability is less pronounced (from 65.85% to 69%), which may indicate that the tasks in this dataset present unique challenges that are not fully mitigated by TOT. This could reflect the complexity or diversity of the problems in HumanEval, suggesting that while TOT improves overall code generation performance, its effectiveness may vary depending on the specific nature of the coding tasks.

Answer to RQ1.1: The Tree of Through (TOT) method enhances ChatGPT semantic consistency and reduces non-determinism, enabling predictable code generation aligned with prompt intentions. Despite varying effectiveness across datasets, TOT improves reliability and accuracy in code generation.

B. RQ 1.2 Syntactic Similarity

The experiments evaluate syntactic similarity using LCS and LED metrics. The first candidate code is used as a reference, and four LCS and LED values are calculated between the reference and four other candidate codes. The average and worst values represent each problem's syntactic similarity. Table III shows the average and worst values of LCS and LED for all problems in a dataset.

TABLE III
RQ 1.2 Syntactic similarity

Syntactic similarity	Metrics	CodeContests	APPS	HumanEval
LCS	Mean	0.96	0.98	0.99
	Worst value	0.96	0.98	0.99
LED	Mean	0.00	0.00	0.00
	Worst value	0.00	0.00	0.00

A high *LCS* signifies that the generated code sequences share long common subsequences, indicating high syntactic similarity between the candidate codes; a zero *LED* value indicates that no modifications are needed to transform one code sequence into another, signifying perfect identity or very close syntactic proximity. This is reflected in our results recorded in Table III. It suggests that not only are the solutions syntactically similar, but they can be identical or nearly identical in many cases. This denotes extreme stability in code generation, with little syntactic variation among the different generated solutions. Comparing our results to those of Ouyang and al. [3], where greater variability was observed, it is clear that the *TOT* has succeeded in minimizing non-determinism in code generation.

answer to RQ1.2: our results show that code generation performance varies across categories, but overall, the similarity values (*LCS*) are high, and the average edit distances (*LED*) are zero, indicating a good match between the generated code sequences and the expected ones, thus demonstrating remarkable performance in terms of syntactic similarity and stability in code generation.

C. RQ1.3 Structural Similarity

Structural similarity measures evaluate the similarity of codes based on their Abstract Syntax Tree (AST). In our experiment, structural similarity is primarily measured using the $pycode_{similar}$ tool with two different configurations, namely $United_{Diff}$ and $Tree_{Diff}$ (more details in Section 1.3). For the five candidate codes for each coding problem, we use the first candidate code as the reference and calculate the structural similarity between the first candidate code and the other four candidates under the $United_{Diff}$ and $Tree_{Diff}$ configurations. Thus, each problem has four values for $United_{Diff}$ and $Tree_{Diff}$ respectively, with each value indicating a measure of structural similarity. We use the average of these four values as well as the worst of them (i.e., the smallest value for $United_{Diff}$ and $Tree_{Diff}$) to represent the structural similarity of each problem. The tableIV shows the average and worst values under the $United_{Diff}$ and $Tree_{Diff}$ configurations for all coding problems in a dataset. the tableIV illustrates our results.

TABLE IV
RQ1.3 STRUCTURAL SIMILARITY RESULTS

Structural similarity	Metrics	CodeContests	APPS	HumanEval
$United_{Diff}$	Mean	0.95	0.98	0.99
	Worst value	0.95	0.98	0.99
$Tree_{Diff}$	Mean	0.95	0.98	0.99
	Worst value	0.95	0.98	0.99

RQ 1.2.3 answer:Our tests show ChatGPT generates very similar code structures (high structural stability). This means the code looks consistent across different problems. This is an improvement over previous findings [3] where code structure varied more. This shows TOT is effective for creating consistent and well-structured code.

D. RQ2: Temperature Influence

The default temperature of ChatGPT is 1. This research question explores whether non-determinism in ChatGPT code generation changes with temperature variations. We use metrics identical to those in RQ1. Due to the limited financial resources (300 $) available for this work, we only present the results for the APPS dataset, with which we were able to experiment with all temperature values. the tableV illustrates the results.

TABLE V
RQ2. INFLUENCE OF TEMPERATURE ON TEST PASS RATE AND OER

Temperature	Test Pass Rate				
	Mean Value	**Mean Variance**	**Mean Max Diff**	**Max Diff**	**Ratio of Worst Cases**
0	0.42	8.41	0.00	1.00	0.11
1	0.18	6.40	0.00	1.00	0.43
2	0.005	0.00	0.00	1.00	0.00
Temperature	**OER**				
	Mean Value	**Worst Value**	**Ratio of Worst Cases**	**–**	**–**
0	1.00	0.00	0.00	–	–
1	1.00	0.00	0.00	–	–
2	1.00	0.00	0.00	–	–
Temperature	**OER (no ex.)**				
	Mean Value	**Mean Variance**	**Mean Max Diff**		
0	0.90	0.00	0.04		
1	0.50	0.00	0.42		
2	0.04	0.00	0.96		
Temperature	**LCS**		**LED**		
	Mean Value	**Worst Value**	**Mean Value**	**Worst Value**	
0	0.98	0.98	0.00	0.00	
1	0.98	0.98	0.00	0.00	
2	0.05	0.05	0.00	0.00	
Temperature	$Unified_{Diff}$		$Tree_{Diff}$		
	Mean Value	**Worst Value**	**Mean Value**	**Worst Value**	
0	0.98	0.98	0.98	0.98	
1	0.98	0.98	0.98	0.98	
2	0.04	0.04	0.04	0.04	

At Temperature=0

- High Performance: at this temperature, the mean values of test pass rate and OER measures indicate high performance and strong similarity in the generated code with little non-determinism. The perfect OER scores highlight complete equivalence in code outputs, while the high values of LCS and $Tree_{Diff}$. $Unified_{Diff}$ and $Tree_{Diff}$ demonstrate strong syntactic and structural similarity.
- Consistency: the consistency of the generated code is confirmed by minimal variance values and low worst-case ratios, indicating reliable and predictable code generation at this temperature (t=0).

At Temperature=1 (Default)

- Performance Decline: Compared to temperature 0, a slight performance drop is observed, with a notable decrease in the mean test pass rate. However, the OER measures remain perfect, and the LCS and $Unified_{Diff}$ and $Tree_{Diff}$ values remain high, suggesting that the quality and similarity of the code stay strong despite increased variability in test results.
- Increase in Non-determinism: an increase in worst-case ratios and maximum variance in test pass rates indicates a slight rise in non-determinism compared to temperature=0.

At Temperature=2

- Drastic Performance Reduction: Temperature=2 leads to a sharp decline in test pass rates with an extremely low average, suggesting that increasing the temperature significantly hinders *ChatGPT*'s ability to generate correct code.
- Uniformity in Performance: However, this temperature shows uniformity in the quality of generated code, as reflected by zero variance values and worst-case ratios. This indicates that, while high temperature generates a variety of responses, it tends towards consistency in the decline in quality.
- Loss of Syntactic and Structural Similarity: LCS and $Tree_{Diff}$ and $Unified_{Diff}$ measures collapse to very low values at this temperature, demonstrating a significant loss of syntactic and structural similarity between the generated codes.

RQ 2 answer: Our results reveal that temperature plays a crucial role in the determinism and quality of code generated by ChatGPT. A temperature of 0 promotes accuracy, similarity, and determinism, while a temperature of 2 leads to excessive creativity that hinders the quality and similarity of the generated code. Contrary to the work of Ouyang and al. [3] which highlights that, contrary to common belief, setting the temperature to 0 does not ensure entirely deterministic behavior of ChatGPT; our results suggest a strong influence of temperature on variability and non-determinism with significantly better performance at reduced temperature, but without completely eliminating non-determinism.

E. RQ 3: Comparing Non-Determinism with Best Candidate Codes in the Same Prediction

RQ1 and RQ2 compare the similarity of five candidate codes generated from multiple requests. Each candidate code is the first candidate from each request. However, ChatGPT can also generate five candidate codes within the same request (the top five candidates ranked by their predictive probabilities). This research question compares the level of non-determinism among candidate codes for the two request configurations mentioned above. Table VI shows the results for the APPS dataset. To simplify the presentation, we use R1 to refer to single requests and R2 to refer to multiple requests. Intuitively, the top five candidates from five different requests (R2) should exhibit more similarity than the top five candidate codes from a single request (R1), as the former are a collection of the best responses for each request and the latter are just the top five. Nevertheless, our observation indicates that the top five candidate codes from a single request (R1) have very similar similarities to the top five candidates from multiple requests (R2). the tableVI illustrates our results.

TABLE VI
RQ3 COMPARING NON-DETERMINISM WITH BEST CANDIDATE CODES IN THE SAME PREDICTION

Request way	Test Pass Rate				
	Mean	Variance	Max diff	Max	Worst case
R1	0.17	0.018	0.16	1.00	0.052
R2	0.18	6.40	0.00	1.00	0.09
Request way	OER				
	Mean	Worst	Worst ratio		
R1	0.57	0.00	0.30		
R2	1.00	0.00	0.00		
Request way	OER (no ex.)				
	Mean	Variance	Max diff		
R1	0.12	0.00	0.80		
R2	0.50	0.00	0.42		
Request way	LCS		LED		
	Mean	Worst	Mean	Worst	
R1	0.15	0.10	60.17	77.42	
R2	0.98	0.98	0.00	0.00	
Request way	$Unified_{Diff}$		$Tree_{Diff}$		
	Mean	Worst	Mean	Worst	
R1	0.48	0.21	0.53	0.26	
R2	0.98	0.98	0.98	0.98	

Interpretation of Results:

- Test Pass Rate: The average pass rates are slightly higher for multiple queries (R2) compared to a single query (R1), suggesting a slight improvement in the quality and relevance of the code generated with *TOT* when using multiple queries. However, the significant variance observed in R2 versus lower variance in R1 indicates a more pronounced non-determinism in the context of multiple queries.
- OER (Output Equivalence Rate): The perfect OER in R2 (1.00) versus a lower OER in R1 (0.57) shows that the codes generated across multiple queries are not only more consistent with each other but also entirely equivalent in terms of output, significantly reducing non-determinism compared to a single query.
- OER (no ex): The difference in the mean OER values without exceptions between R1 and R2 (0.12 vs. 0.5) further highlights the effectiveness of *ToT* in producing functionally similar codes, particularly in the context of multiple queries.
- LCS (Longest Common Subsequence) and LED (Levenshtein Edit Distance): The results show a dramatic improvement in syntactic similarity in R2 (LCS at 0.98 and LED at 0.00), indicating that the codes generated in multiple queries are almost identical, in stark contrast to the low LCS values and high LED values observed in R1.
- $Unified_{Diff}$ and $Tree_{Diff}$: The high values for these measures in R2 (0.98) versus moderate values in R1 (around 0.5) demonstrate that *TOT* significantly enhances

the structural similarity of the generated codes in the context of multiple queries, making the codes not only functionally but also structurally similar.

RQ3 answer: Our results with the "Tree of Through" (TOT) method have revealed that this approach significantly improves the determinism and similarity of codes generated by ChatGPT, especially in the context of multiple queries (R2) compared to a single query (R1). The results show superior quality and increased consistency of codes in R2, with significantly improved test success rates, output equivalence rates (OER), and syntactic (LCS) and structural ($Unified_Diff$ and $Tree_Diff$) similarity measures. This observation highlights the effectiveness of TOT in guiding ChatGPT towards producing more consistent and accurate responses.

VII. Conclusion

This research explored and highlighted the potential of the *Tree of Thoughts* (ToT) approach to mitigate the issue of non-determinism in large language models (LLM) within the specific context of code generation. We demonstrated that this approach, by structuring the models' reasoning around possible paths leading to a solution, promotes more reproducible and coherent outcomes. By translating classical problem-solving concepts into actionable methods for contemporary language models, the ToT approach proves particularly suitable for tasks where *accuracy and reliability are critical*, such as code generation. This research also emphasizes the importance of systematically evaluating the reproducibility of techniques applied to LLMs, an often-overlooked yet critical aspect for their adoption in production environments. Finally, this intersection between classical artificial intelligence techniques and the modern capabilities of LLMs opens promising avenues for future research. It has the potential to transform fields such as assisted programming, automated decision-making, and other applications where the combination of intuition and analytical thinking contributes to more robust and reliable solutions.

References

[1] J. Liu, A. Liu, X. Lu, S. Welleck, P. West, R. L. Bras, Y. Choi, and H. Hajishirzi, "Generated knowledge prompting for commonsense reasoning," *arXiv preprint arXiv:2110.08387*, 2021.

[2] X. Wang, J. Wei, D. Schuurmans, Q. Le, E. Chi, S. Narang, A. Chowdhery, and D. Zhou, "Self-consistency improves chain of thought reasoning in language models," *arXiv preprint arXiv:2203.11171*, 2022.

[3] S. Ouyang, J. M. Zhang, M. Harman, and M. Wang, "Llm is like a box of chocolates: the non-determinism of chatgpt in code generation," *arXiv preprint arXiv:2308.02828*, 2023.

[4] M. Lee, P. Liang, and Q. Yang, "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*, 2022, pp. 1–19.

[5] S. Gulwani, "Automating string processing in spreadsheets using input-output examples," *ACM Sigplan Notices*, vol. 46, no. 1, pp. 317–330, 2011.

[6] S. Chatterjee, D. Saha, A. Sharma, and Y. Verma, "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*, pp. 1–21, 2022.

[7] S. Yao, D. Yu, J. Zhao, I. Shafran, T. L. Griffiths, Y. Cao, and K. Narasimhan, "Tree of thoughts: Deliberate problem solving with large language models," *arXiv preprint arXiv:2305.10601*, 2023.

[8] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[9] I. I. Revzin and Y. Gentilhomme, "¡¡ les¿¿ modèles linguistiques," 2020.

[10] W. Ling, E. Grefenstette, K. M. Hermann, T. Kočiskỳ, A. Senior, F. Wang, and P. Blunsom, "Latent predictor networks for code generation," *arXiv preprint arXiv:1603.06744*, 2016.

[11] T. B. Hashimoto, K. Guu, Y. Oren, and P. S. Liang, "A retrieve-and-edit framework for predicting structured outputs," *Advances in Neural Information Processing Systems*, vol. 31, 2018.

[12] L. Dong and M. Lapata, "Language to logical form with neural attention," *arXiv preprint arXiv:1601.01280*, 2016.

[13] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[14] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever *et al.*, "Improving language understanding by generative pre-training," 2018.

[15] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang *et al.*, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *arXiv preprint arXiv:2102.04664*, 2021.

[16] J. Long, "Large language model guided tree-of-thought," *arXiv preprint arXiv:2305.08291*, 2023.

[17] X. Chen, M. Lin, N. Schärli, and D. Zhou, "Teaching large language models to self-debug," *arXiv preprint arXiv:2304.05128*, 2023.

[18] G. Kim, P. Baldi, and S. McAleer, "Language models can solve computer tasks," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[19] I. Schlag, S. Sukhbaatar, A. Celikyilmaz, W.-t. Yih, J. Weston, J. Schmidhuber, and X. Li, "Large language model programs," *arXiv preprint arXiv:2305.05364*, 2023.

[20] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le *et al.*, "Program synthesis with large language models," *arXiv preprint arXiv:2108.07732*, 2021.

[21] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago *et al.*, "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.

[22] D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, A. Arora, E. Guo, C. Burns, S. Puranik, H. He, D. Song *et al.*, "Measuring coding challenge competence with apps," *arXiv preprint arXiv:2105.09938*, 2021.