Received: 27 August 2023 | Revised: 22 November 2023 | Accepted: 20 December 2023

DOI: 10.4218/etrij.2023-0357

SPECIAL ISSUE

ETRI Journal WILEY

Framework for evaluating code generation ability of large language models

Sangyeop Yeo[1] | **Yu-Seung Ma**[1,2] | **Sang Cheol Kim**[2] | **Hyungkook Jun**[2] | **Taeho Kim**[2]

[1]Division of Artificial Intelligence, University of Science and Technology, Daejeon, Republic of Korea

[2]Artificial Intelligence Computing Research Laboratory, Electronics and Telecommunications Research Institute, Daejeon, Republic of Korea

Correspondence
Yu-Seung Ma, Electronics and Telecommunications Research Institute, Daejeon, Republic of Korea.
Email: ysma@etri.re.kr

Funding information
This work was supported by an Institute of Information & Communications Technology Planning & Evaluation (IITP) grant (2022-0-00995, automated reliable source code generation from natural language descriptions, 95%) and a National Research Council of Science & Technology (NST) grant (Global-23-001, SeCode: Collaborative intelligent model for secure program code generator, 5%) funded by the Korea government (MSIT).

Abstract

Large language models (LLMs) have revolutionized various applications in natural language processing and exhibited proficiency in generating programming code. We propose a framework for evaluating the code generation ability of LLMs and introduce a new metric, *pass-ratio@n*, which captures the granularity of accuracy according to the pass rate of test cases. The framework is intended to be fully automatic to handle the repetitive work involved in generating prompts, conducting inferences, and executing the generated codes. A preliminary evaluation focusing on the prompt detail, problem publication date, and difficulty level demonstrates the successful integration of our framework with the LeetCode coding platform and highlights the applicability of the *pass-ratio@n* metric.

KEYWORDS
code generation, evaluation metric, large language model, natural language processing, software engineering

1 | INTRODUCTION

Recently, large language models (LLMs) have emerged as powerful tools for natural language processing [1–3], revolutionizing various applications such as text generation, translation, and question answering. Although initially intended for natural language understanding and generation, LLMs have also displayed remarkable proficiency in comprehending and generating programming code. Employing LLMs to generate source code implies utilizing advanced machine-learning models to assist in implementing code for specific tasks or functions. Several LLMs, such as Codex, AlphaCode, and CodeGen [4–6], are available for generating source code. These models have been trained on extensive datasets comprising both source code and textual descriptions from various programming languages to provide coding solutions, fix syntax errors, and even help designing algorithms.

Software developers increasingly use LLMs to facilitate code generation. By simply providing a natural language description outlining the desired functionality, LLMs generate the corresponding code. LLMs that generate code can boost developer efficiency, assist with rapid prototyping, ensure that the code implements specific

rules, reduce mistakes, and assist nonexperts. Although LLMs offer these advantages, their performance should be further improved, and a thorough quality evaluation is required. In fact, the generated code must be rigorously reviewed and tested to ensure that it meets the design requirements and is error-free.

No general accepted guidelines for evaluating the quality of source code generated by LLMs are available. LLMs are evaluated using diverse metrics and data, thus undermining consistency and hindering comparisons of different models and techniques. Moreover, inconsistent evaluations impede understanding the effectiveness and reliability of LLM-based code generation.

BLEU [7], CodeBLEU [8], and *pass@k* [9] are often used to assess code functionality. However, the BLEU and CodeBLEU metrics evaluate codes based on their syntactic similarity to a specific single answer. Consequently, even if an LLM generates functionally correct code, it may receive a low score if its syntax differs from that of the reference solution. Alternatively, the *pass@k* metric assesses functionality based on the actual execution results and may thus be more appropriate for evaluating codes. The *pass@k* metric evaluates the generated code by executing it and checking whether all the test cases are passed. However, this binary approach, which solely assesses whether the code is completely correct, hinders quality evaluation at a more granular level. This limitation underscores the necessity of introducing additional metrics to capture varying degrees of accuracy.

The selection of datasets for evaluating code generation is also important. Several studies [10–12] have been proposed to evaluate the quality of LLMs for code generation. However, the datasets used in the experiments varied in terms of program size, complexity, and format. For example, some datasets consist of specifications and codes that are only a few lines long and relatively straightforward, whereas others encompass complex problems found in coding challenges. Additionally, they coding problems may had been included during LLM training. Hence, datasets should be carefully selected to ensure realistic and unbiased assessment of the models. Therefore, data selection guidelines should be established.

To address the above-mentioned problems, we propose a systematic framework for evaluating LLMs with an emphasis on functionality. Nonfunctional evaluations, such as readability and complexity, have been extensively studied [10–13]. Our framework improves previous research by addressing non-functionality. We first describe factors for dataset selection. We then propose a process for evaluating an LLM with a dataset that meets specific criteria. In addition, we address the limitations of the existing evaluation methods by introducing a new metric, *pass-ratio@n*, which captures the multifaceted nature of code quality. Because evaluating LLMs requires considerable repetitive work, from generating queries, making inferences, and executing the generated codes, we introduce a fully automatic process.

The contributions of this study are as follows:

- We propose an evaluation framework to assess the code generation ability of LLMs. The framework emphasizes integration with common coding platforms.
- We derive a new metric, *pass-ratio@n*, which provides a more granular measure of accuracy by considering the pass rate of test cases across *n* inferences.
- We conducted a preliminary evaluation of the proposed framework to demonstrate its fully automatic capabilities.

The remainder of this paper is organized as follows. Section 2 presents related work on metrics and datasets. Section 3 outlines the dataset conditions for LLM evaluation. Section 4 presents an evaluation framework that includes the new metric. Section 5 describes the proposed framework. Section 6 discusses the limitations, and Section 7 presents our conclusions.

2 | RELATED WORK

2.1 | Metric pass@k

The *pass@k* [4, 9] metric allows to assess the code generation ability of an LLM based on the code execution results. Given a coding problem, the *pass@k* metric involves analyzing k different code solutions generated by the LLM. If at least one of the k generated solutions passes all the tests, the LLM is considered to have solved the problem.

The *pass@k* metric introduced in Kulal et al. [9] was further refined in Chen et al. [4] to consider additional n solutions, with $n > k$. The Codex model was evaluated using the *pass@k* metric in Chen et al. [4]. The *pass@k* metric is expressed as

$$pass@k := \mathop{\mathbb{E}}_{\text{problems}}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right], \tag{1}$$

where c represents the number of correct code solutions. To obtain functional correctness according to the *pass@k* metric, the code should pass all the tests, resembling validation by human programmers of real-world code.

However, perfectly passing all the tests for a given coding problem is challenging not only for LLMs but also for humans. While examining test success is important, it is also valuable to consider the proportion of test cases that have been successfully passed.

2.2 | Datasets

Selecting appropriate datasets is essential for properly assessing the performance of LLMs. Methods that evaluate the code generation ability of LLMs based on code execution, such as the *pass*@*k* metric, depend the quality of test data. Common datasets for these types of assessments include HumanEval [4] and MBPP [14]. HumanEval [4] contains 164 handwritten Python problems, each accompanied by a function signature, descriptions, and multiple unit tests. MBPP [14] comprises 1000 crowdsourced Python problems, each with task descriptions, code solutions, and three test cases. Their multilingual versions for code generative models, HumanEval-X [15] and MBXP [16]), have also been developed in recent projects.

The abovementioned datasets typically involve relatively simple programming problems. For example, they involve problems with a source code of fewer than 200 characters and approximately 10 lines. Evaluating LLMs using simple coding problems may result in inflated and unrealistic *pass*@*k* scores. In Roziére [17], GPT-4 solved 67% of the problems with only a single inference (that is, *pass*@1), and the CodeLlama-34B model achieved a 53.7% *pass*@1 score and 88.2% *pass*@100 score. A high *pass*@100 value implies a high possibility that the LLM can find a solution within 100 attempts. However, it is impractical to perform several inferences for a single coding problem because it is resource-intensive, and the most appropriate solution among all the possible answers should be selected.

More realistic problems are being used from coding platforms, such as Codeforces and LeetCode. For example, the APPS dataset [18] includes 10,000 real-world problems from various open-access coding websites, such as Codeforces and Kattis, with 131,836 test cases. LeetCode problems have often been used for evaluations in recent studies [19, 20]. The LeetCode platform provides useful information for each problem including problem description, topic category, difficulty level, input/output examples, publication date, and massive test data. However, the information available in LeetCode has not been fully exploited when evaluating code generation using LLMs [19, 20]. In previous studies, LLMs were mainly analyzed focusing on the language type or difficulty level. Hence, various challenges remain for fully using the rich information provided by coding platforms such as LeetCode and automating evaluation, which is essential for assessing the real-world programming ability of LLMs.

3 | FACTORS FOR DATASET SELECTION

To evaluate the code generation ability of LLMs, various datasets encompassing a wide range of real-world coding problems should be used. Previous studies [4, 9, 17, 20] have often assessed LLMs from a limited perspective, mainly focusing on particular programming languages or difficulty levels.

However, additional factors must be considered when evaluating the code generation ability of LLMs. For instance, the temporal context involves determining whether coding problems are created before or after training the model. This helps ensure that the model does not directly learn the solutions or is exposed to similar problems during training. Real-world relevance is another factor to filter overly simple and outdated coding problems. Furthermore, the availability of the test data is essential to verify the functionality of generated code. Hence, we evaluate code functionality based on the execution results of actual test cases. The amount of test data is important because data scarcity impedes ensuring code correctness. Diverse test data ensure that the code solution is not skewed toward specific inputs or scenarios and allows to determine whether the LLM correctly solves a coding problem. However, most existing datasets lack a comprehensive set of test cases.

Platforms such as LeetCode can help handling the factors for properly evaluating the coding ability. LeetCode is a popular online platform used to solve coding problems and provides a vast collection of problems. Moreover, each coding problem contains representative information such as the difficulty level and topic. Considering the abovementioned factors, the advantages of using LeetCode are as follows. First, LeetCode regularly hosts coding challenges and adds up-to-date problems. This allows to assess an LLM using coding problems created after model training. In fact, using coding problems published after the LLM release allows to evaluate the LLM ability to handle new and potentially unseen challenges. Second, the coding problems in LeetCode are real-world coding interview questions and challenges faced in actual software development scenarios. Finally, it provides a test dataset for each coding problem, often comprising more than 100 test cases, enabling a deep evaluation of the quality of generated code.

4 | PROPOSED FRAMEWORK AND METRIC

4.1 | Framework architecture

We propose a framework for assessing the code generation ability of LLMs in alignment with platforms such as LeetCode. The proposed framework is expected to support various metrics, and the entire evaluation process is automated.

Figure 1 shows the architecture of the proposed evaluation framework. The architecture is divided into three stages: i) coding problem preparation, ii) LLM execution, and iii) code analysis. These stages proceed sequentially, and storing the outcomes of the previous stage allows skipping it and proceeding to the next stage.

(1) Coding problem preparation

Evaluating the code generation ability of LLMs generally involves determining the number of given coding problems that are solved using the generated codes. However, this approach often overlooks the multifaceted analysis of coding problems. For instance, it is meaningful to understand the influence of accuracy with adequate granularity of the prompt input on the LLM for solving a coding problem and evaluate the output variability based on the prompt details. Another factor to consider for a coding problem is the publication date. This temporal information is helpful in ensuring that the problems are not part of the LLM training data.

Table 1 presents the information that must be extracted during coding problem preparation. The problems are designed using the LeetCode platform. Items such as problem descriptions, execution examples, and constraints are used to generate prompts with varying levels of detail. Aspects such as difficulty level, topic, and publication date are used to analyze the evaluation results from various perspectives. Finally, the signature is required to set up an execution environment to run the test cases.

LeetCode offers a graphical user interface that allows users to handle coding problems individually. However, manually extracting coding problems by individually clicking on each problem and copying and pasting the description is labor-intensive. To evaluate LLMs using several coding problems, automating problem extraction is crucial. Our implementation automatically extracts the

TABLE 1 Information to be extracted from every coding problem.

Item	Description
Problem description	Content of the task to be solved through programming. This typically encompasses the purpose of the problem.
Execution examples	Sample inputs and expected outputs.
Constraints	Limitations, rules, or conditions that must be followed. They cover input ranges, sizes, or other restrictions.
Difficulty level	Coding complexity of a problem. Many platforms, including LeetCode, categorize problems into three difficulty levels: easy, medium, and hard.
Topics	Specific category or area of algorithms, data structures, and programming concepts. This categorization enables analysis based on specific domains.
Publication date	Publication date of a coding problem. Using problems published after training confirms that the evaluated LLM does not have pre-existing knowledge.
Signature	Structure of a function, encompassing the function name, parameter types, and return types.

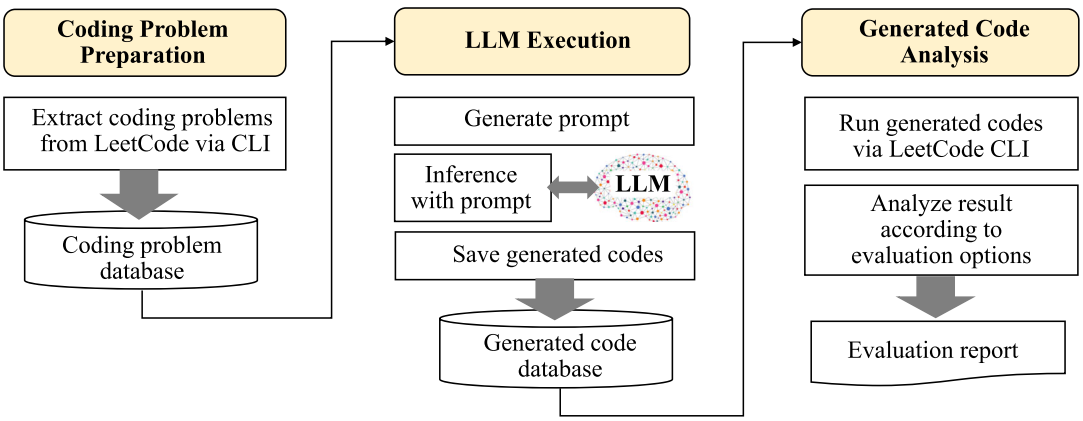

FIGURE 1 Overview of LLM evaluation framework using LeetCode.

information listed in Table 1 by leveraging third-party open-source command line interface (CLI) libraries for LeetCode.

(2) LLM execution

This stage consists of three steps: generating prompts, performing inferences with prompts, and saving the generated codes.

- **Generate prompt**: Referring to Table 1, this step creates prompts of different depths: (i) only problem description, (ii) adding constraints, and (iii) adding execution examples. By using prompts with varying levels of detail, we can examine the influence of the prompts and determine those that produce the best results. The function signature is also included to produce a code appropriate to the structure.
- **Infer from prompt**: In this step, the LLM generates the intended code. We feed the prompts created in the previous step to the LLM and obtain the source code produced as an inference result. Inference can be performed using any kind of LLM, including in-house, downloaded, and commercially available models.
- **Save generated codes**: The LLM execution phase generates code using several prompts, resulting in multiple problem–prompt pairs, and several inference iterations are performed per pair. We store the results after each inference to avoid future re-executions and reuse prior work. The stored source code instances are augmented with metadata including the corresponding coding problem, prompt details, and target programming language, facilitating future performance analysis. When storing, outliers are filtered.

(3) Generated code analysis

This stage comprises two steps: (i) running the generated codes with test cases and (ii) analyzing the results according to the metrics.

- **Run generated codes with test cases**: To verify the functional correctness of the generated code, it should be executed for test cases. LeetCode provides a test execution environment. After submitting the source code on the platform and activating the test process, it shows the test results, which include the total number of test cases executed, count of successful cases, and detailed insights on runtime and memory consumption. Our implementation automates these repetitive processes by using the corresponding LeetCode CLI library.
- **Analyze results according to metrics**: Codes generated by the LLM are evaluated using predetermined metrics that reflect aspects such as accuracy, efficiency, and readability. Leveraging the information extracted alongside coding problems allows to analyze the LLM capabilities from various perspectives. For example, analyses based on criteria such as problem complexity, publication date, topic, and level of prompt detail are possible. This method not only offers insights into the LLM abilities but also identifies areas for improvement, possibly guiding future refinements.

4.2 | Metric *pass-ratio@n*

The *pass@k* metric is widely used to evaluate the effectiveness of code generated by LLMs based on the execution results. However, this metric considers the code to be either completely correct or incorrect using a binary pass/fail criterion. Consequently, varying degrees of correctness cannot be considered.

Consider a simple example involving a coding problem with 10 test cases. Let us focus on the *pass@1* metric, which is the simplest form of *pass@k*. For *pass@1*, an LLM-generated code solution receives a score of 1 only if it passed all 10 tests. If it fails in the 10 test cases, the score is 0. Even if a code passes 9 out of the 10 test cases, the score is still 0, as if it had passed none. This binary scoring system fails to acknowledge for partial success in the test cases. This limitation remains even when $k > 1$. Consequently, *pass@k* lacks the granularity to distinguish nearly correct from entirely incorrect solutions. This scoring limitation emphasizes the need for more refined metrics to evaluate LLM code generation. Thus, we propose a new metric, *pass-ratio@n*. The proposed *pass-ratio* is the proportion of passed tests. Considering the probabilistic nature of LLMs, which may not produce identical code solutions across inferences, we perform n inferences. The average pass ratio value across the n solutions is then used to calculate the score, thereby mitigating the bias from a single inference.

For a given solution i $(0 < i \leq n)$, $\textit{pass-ratio}_i$ is calculated by squaring the ratio obtained by dividing the number of passed test cases for code i by the total number of test cases. By squaring the value, solutions with higher pass ratios are assigned more weight, thus reflecting a higher degree of accuracy.

$$\textit{pass-ratio}_i = \left(\frac{\text{the number of passed test cases at code } i}{\text{the number of test cases}}\right)^2.$$

The value of *pass-ratio@n* represents the average *pass-ratio* across n generated codes:

$$\textit{pass-ratio@n} = \frac{\sum_{i=1}^{n} \textit{pass-ratio}_i}{n}.$$

5 | FRAMEWORK EVALUATION

We analyzed the proposed framework to ensure that the LLM evaluation was fully automatic and effective. This analysis focused on the feasibility of the proposed framework and metrics rather than on an exhaustive analysis of the LLM performance. We chose GPT-3.5 and GPT-4 as the LLMs to be evaluated and integrated them with the commercial application programming interface (API) provided by OpenAI to access their LLMs.

We conducted analyses to evaluate the LLM prompt details, problem publication date, and proposed metric.

5.1 | Analysis of prompt details

We first explored the impact of the prompt details on code generation. We assumed that more details increased the accuracy of code generation. To confirm this hypothesis, we devised three types of prompts with varying levels of detail:

- (Type 1) Basic problem description: This is a simple straightforward description of the coding problem.
- (Type 2) Problem description with constraints: An extended version of the type 1 description including constraints imposed to the solution.
- (Type 3) Problem description with constraints and examples: This is the most detailed type of query and encompasses both constraints and illustrative examples.

The problem type thus ranged from the least (i) to the most (iii) detailed. For the analysis, we randomly selected 10 coding problems from LeetCode. The three types of prompts were created per coding problem, thus evaluating 30 queries (10 problems × 3 queries). We generated source codes for the 30 queries using GPT-3.5 and measured the code accuracy using LeetCode. Table 2 lists the results of whether the generated codes passed the tests. In this analysis, the source code was generated once by a single inference per prompt.

Table 2 shows that the codes generated using prompts of types 1 and 2 solved 6 and 7 out of the 10 problems, respectively, while those of type 3 solved 8, indicating the best performance among the three types of prompts.

For a deeper analysis, we present a description of coding problem 4 in Table 2. This problem was retrieved from LeetCode.

> (Source https://leetcode.com/problems/maximum-matching-of-players-with-trainers/).
> **Description of coding problem 4** Title: Maximum matching of players with trainers You are given a 0-indexed integer array players, where players[i] represents the ability of the *i*th player. You are also given a 0-indexed integer array trainers, where trainers[j] represents the training capacity of the *j*th trainer. The *i*th player can match with the *j*th trainer if the player's ability is less than or equal to the trainer's training capacity. Additionally, the *i*th player can be matched with at most one trainer, and the *j*th trainer can be matched with at most one player. Return the maximum number of matchings between players and trainers that satisfy these conditions.

For problem 4, the source code generated by GPT-3.5 using the three types of prompts is shown in Figure 2. Among the three prompts, only the code generated with the type 3 prompt passed all the test cases. The code generated with the type 2 prompt was implemented similarly to that generated with type 3. However, the code from the type 2 prompt failed to pass the test case with input (`players=[1,1000000000]`, `trainers=[1000000000,1]`). Although the expected output for the test case was 2, its output was 1. This originated from the fourth line of the code, in which an incorrect parameter value was used. Changing the code from

TABLE 2 Pass/fail results of source codes generated by GPT-3.5 for 10 LeetCode coding problems.

	Coding problem									
Prompt type	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Type 1 (description only)	Pass	Pass	Pass	**Fail**	**Fail**	**Fail**	Pass	Pass	Pass	**Fail**
Type 2 (description + constraints)	Pass	Pass	Pass	**Fail**	**Fail**	Pass	Pass	Pass	Pass	**Fail**
Type 3 (description + constraints + example)	Pass	Pass	Pass	Pass	**Fail**	Pass	Pass	Pass	Pass	**Fail**

[Description]

```
class Solution:
    def matchPlayersAndTrainers(self, players, trainers):
        matchings = 0
        for player in players:
            for i in range(len(trainers)):
                if players[player] <= trainers[i]:
                    matchings += 1
                    trainers.pop(i)
                    break
        return matchings
```

[Description + constraints]

```
class Solution:
    def matchPlayersAndTrainers(self, players, trainers):
        players.sort()  # sort the players in ascending order of ability
        trainers.sort(reverse=True)
        num_matchings = 0  # to keep track of the number of matchings
        i = 0  # pointer for players
        j = 0  # pointer for trainers
        while i < len(players) and j < len(trainers):
            if players[i] <= trainers[j]:
                num_matchings += 1  # increment the count of matchings
                i += 1  # move to the next player
                j += 1  # move to the next trainer
            else:
                j += 1  # move to the next trainer
        return num_matchings
```

[Description + constraints + example]

```
class Solution:
    def matchPlayersAndTrainers(self, players, trainers):
        players.sort()
        trainers.sort()
        matchings = 0
        player_index = 0
        trainer_index = 0
        while player_index < len(players) and trainer_index < len(trainers):
            if players[player_index] <= trainers[trainer_index]:
                matchings += 1
                player_index += 1
                trainer_index += 1
            else:
                trainer_index += 1
        return matchings
```

FIGURE 2 Comparison of generated codes according to prompt details.

'`trainers.sort(reverse=True)`' to '`trainers.sort()`' corrected the error.

While this analysis was based on a small sample of 10 coding problems, our initial observations suggest that there is a correlation between the level of detail in the prompt and accuracy of the source code generated by the LLM.

5.2 | Analysis of publication date

We also designed an experiment to evaluate the source code generation ability of the LLM according to the publication year of the coding problem. Coding problems published before the LLM training date are probably easy for the model to solve, but those created after model training may be more difficult.

To investigate this assumption, we organized the coding problems according to the publication date. Considering that GPT-3.5 was initially released on March 15, 2022, we chose coding problems from 2023, 2022, and up to 2021 for evaluation. We also employed an additional classification based on the difficulty levels of easy, medium, and hard. For each year and difficulty level, we randomly selected 10 coding problems and conducted five inferences per problem using GPT-3.5, resulting in an examination of five generated code solutions per problem.

Table 3 lists the number of coding problems successfully solved (i.e., every test case in a problem passed) out of a total of 10 problems per group.

Table 3 shows that GPT-3.5 performs well on coding problems published until 2021, solving eight easy, four medium-difficulty, and three hard problems. However, its performance declines when confronted with problems from 2022 and 2023. For 2022 and 2023, some easy and medium-difficulty problems were solved, but no difficult problem was solved.

While GPT demonstrated proficiency in solving problems up to 2021, it encountered difficulties with problems that emerged after its training date. The decline in performance for problems from 2022 and 2023 may be attributed to the introduction of new contexts or topics not included in the training data. However, a true understanding of natural language specifications requires the ability to handle evolving problems. This shows the fundamental difference between human cognition and current LLMs. Humans can infer, extrapolate, and make educated guesses about unfamiliar topics based on prior knowledge and experience. Although models like GPT-3.5 can attempt similar tasks using patterns extracted from their vast datasets, they are inherently limited by existing knowledge. Consequently, they cannot genuinely understand or reason for completely new topics or contexts without sufficient reference in their training data. This underscores the importance of iterative training and model updates in artificial intelligence systems.

TABLE 3 Solved problems out of 10 randomly selected coding problems categorized by publication year and difficulty level using GPT-3.5 with five inferences per problem.

Difficulty level	Publication year		
	~2021	2022	2023
Easy	8	6	6
Medium	4	2	1
Hard	3	0	0

5.3 | Analysis of *pass-ratio@n* metric

We first evaluated the *pass-ratio@n* calculation using three example coding problems in comparison with *pass@k*. We generated five coding solutions per problem using LLM inference. Table 4 lists the number of passed test cases and total number of test cases per LLM-generated code solution. Given that five solutions were generated, we set n to 5 for the *pass-ratio@n* metric. Similarly, we configured n to 5 in *pass@k* while varying k from 1 to 5 to facilitate a comprehensive comparison with *pass-ratio@5*.

Table 4 lists the *pass@k* ($k = 1$–5, $n = 5$) and *pass-ratio@n* ($n = 5$) scores. For the first coding problem, the five generated solutions passed every test case, resulting in *pass@k* ($k = 1$–5, $n = 5$) and *pass-ratio@5* with values of 100%. For the second coding problem, the solutions for problems 2, 3, and 5 passed all the tests, while those for problems 1 and 4 failed to pass some of the 135 test cases, ultimately receiving a fail classification. As three of the five solutions passed, the *pass@1* score was 60%. For *pass@5*, because at least one of the five solutions passed, the result was 100%. The *pass-ratio@5*, calculated as the average of the squared pass rates of each solution, was 78%. In the third coding problem, the five solutions failed, resulting in *pass@k* ($k = 1$–5, $n = 5$) of 0%. However, the generated solutions passed a high portion of the 61 test cases, suggesting that they were effective. Only *pass-ratio@5* reflected this phenomenon. In fact, *pass-ratio@5* was 61% for the third problem, reflecting the test pass rate.

We then integrated our framework with the LeetCode platform to examine the applicability of the *pass-ratio@n* metric. This analysis involved calculating and comparing *pass@1*, *pass@5*, and *pass-ratio@5* for GPT-3.5 considering the publication date and difficulty of the coding

TABLE 4 Comparison of *pass@k (k = 1–5, n = 5)* and *pass-ratio@5* for three coding problems.

Coding problem		1	2	3
Solution	1	72/72 (pass)	128/135 (fail)	56/61 (fail)
	2	72/72 (pass)	135/135 (pass)	48/61 (fail)
	3	72/72 (pass)	135/135 (pass)	21/61 (fail)
	4	72/72 (pass)	0/135 (fail)	48/61 (fail)
	5	72/72 (pass)	135/135 (pass)	56/61 (fail)
pass@1 (n = 5)		100%	60%	0%
pass@2 (n = 5)		100%	90%	0%
pass@3 (n = 5)		100%	100%	0%
pass@4 (n = 5)		100%	100%	0%
pass@5 (n = 5)		100%	100%	0%
pass-ratio@5		100%	78%	61%

TABLE 5 Results of *pass@1, pass@5*, and *pass-ratio@5* using GPT-3.5 for 10 coding problems in various categories.

Difficulty level	Year	*pass@1* (*n* = 5)	*pass@5* (*n* = 5)	*pass-ratio@5*
Easy	~2021	80%	90%	80%
	2022	62%	80%	72%
	2023	64%	80%	70%
Medium	~2021	40%	60%	63%
	2022	16%	20%	33%
	2023	14%	30%	25%
Hard	~2021	26%	40%	33%
	2022	0%	0%	13%
	2023	0%	0%	17%

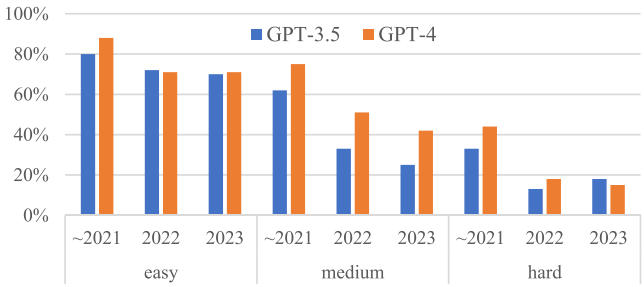

FIGURE 3 Results of *pass-ratio@5* for GPT-3.5 and GPT-4 according to publication year and difficulty level.

problems. The *n* value for *pass-ratio@n* was set to 5, which implied five inference attempts per problem. For each category, 10 coding problems were randomly chosen based on the publication year and difficulty level, resulting in 50 inferences per category. For the three difficulty levels (easy, medium, and hard), this number was multiplied by 3, resulting in 150 inferences. Finally, the publication years up to 2021, 2022, and 2023, multiplied the required inferences by 3, resulting in a total of 450 inferences. This task was labor-intensive, but the proposed framework automated the repetitive processes.

Table 5 lists the *pass@1*, *pass@5*, and *pass-ratio@5* scores for GPT-3.5 across 10 randomly selected problems in each category. For the easy problems, all the metrics showed high scores exceeding 60%. For the medium difficulty, the scores drastically dropped for problems published after 2022. This trend was more pronounced for the hard problems. Notably, the *pass*@1 and *pass*@5 metrics recorded scores of 0 in 2022 and 2023, indicating that no problems were solved. Nevertheless, as inferred from the values of the proposed *pass-ratio@5*, GPT-3.5 managed to pass some of the test cases, indicating certain problem-solving ability. In contrast, the *pass*@*k* metrics failed to capture the potential problem-solving ability of the model.

Finally, we extended our evaluation to include GPT-4 by applying the same method to calculate *pass-ratio@5* as for GPT-3.5. Figure 3 shows the *pass-ratio@5* scores for the GPT-3.5 and GPT-4 LLMs. Both models performed relatively well for problems dating up to 2021. For such problems, a *pass-ratio@5* value of over 70% indicated a commendable problem-solving ability. However, *pass-ratio@5* dropped to less than half for problems after 2021, and this decline was more pronounced for hard problems. GPT-4 generally exhibited a higher *pass-ratio@5* than GPT-3.5. When analyzing the trends of these values across the different categories for each model, similar trends were observed for GPT-3.5 and GPT-4.

6 | LIMITATIONS AND DISCUSSION

For the framework evaluation in Section 5, we selected 10 coding problems from different publication years and difficulty levels. Although we performed 450 inferences across three years, three difficulty levels, and five repetitions, the number of inferences was insufficient to generalize the performance of the LLM. To address this limitation, future studies should incorporate a more extensive group of coding problems. Nevertheless, the proposed framework seems to provide efficiency with time and cost savings by automating the evaluation of LLMs.

The performance of the proposed *pass-ratio@n* method may be affected by the value of n. When $n = 1$, the performance evaluation is based on a single inference. The *pass-ratio@1* value is adequate under low randomness (for example, low temperature in an LLM) because it often yields identical code solutions for the same input. However, under higher randomness, this approach may not be reliable because of the increased variability in the generated codes. In such cases, increasing n can improve the generalization of the evaluation, although this requires multiple inferences and executes the generated code solutions for all the test data, which can be resource-intensive.

Using the *pass-ratio@n* metric, the LLM ability for code generation is evaluated based on the pass rate of test cases. Therefore, the effectiveness of this metric is influenced by the quality of the test cases. This aspect becomes important because the number and complexity of test cases vary across coding problems. Thus, the adopted real-world coding platforms should be reputable to ensure credibility of the findings.

Our framework evaluates LLMs and can use coding platforms, such as LeetCode. However, this approach requires integration of an API that facilitates communication with these platforms through a CLI. In addition, we must adhere to the licensing agreements of the data providers and use the problem data appropriately.

7 | CONCLUSION

We introduce a framework for systematically evaluating the code generation ability of LLMs. We first analyze essential factors to consider in dataset selection and determine that coding platforms such as LeetCode can adequately cover such factors. The proposed framework is intended to be fully automated to manage the repetitive processes involved in generating queries, conducting inferences, and executing the generated codes. In addition, a new metric, *pass-ratio@n*, is introduced to measure accuracy with granularity by considering the test pass rate.

Our preliminary experimental results indicated that the prompt details affected the quality of the generated source code, and the targeted model was less effective at solving coding problems published after the date of LLM training. In addition, the *pass-ratio@n* metric could successfully measure the closeness of the generated code to being complete and functional, representing a nuanced and useful way to assess the performance of LLMs in code generation. This study was aimed solely to confirm the applicability of the proposed framework using a small number of problems and generated code. To generalize our findings, a systematic analysis with a larger dataset is necessary. Future studies will involve refining the framework and conducting a comprehensive evaluation of various LLMs using the proposed framework and metric.

CONFLICT OF INTEREST STATEMENT

The authors declare that there are no conflicts of interest.

ORCID

Yu-Seung Ma https://orcid.org/0000-0002-4168-5515
Sang Cheol Kim https://orcid.org/0000-0002-1925-2588
Taeho Kim https://orcid.org/0000-0002-5061-206X

REFERENCES

1. M. M. Abdollah Pour and S. Momtazi, *Comparative study of text representation and learning for persian named entity recognition*, ETRI J. **44** (2022), no. 5, 794–804. DOI 10.4218/etrij.2021-0269
2. C. Park, J. Lim, J. Ryu, H. Kim, and C. Lee, *Simple and effective neural coreference resolution for korean language*, ETRI J. **43** (2021), no. 6, 1038–1048. DOI 10.4218/etrij.2020-0282
3. A. Prakash, N. K. Singh, and S. K. Saha, *Automatic extraction of similar poetry for study of literary texts: An experiment on hindi poetry*, ETRI J. **44** (2022), no. 3, 413–425. DOI 10.4218/etrij.2019-0396
4. M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, *Evaluating large language models trained on code*, arXiv preprint, 2021. DOI 10.48550/arXiv.2107.03374

5. Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. D. Lago, T. Hubert, P. Choy, C. de Masson d'Autume, I. Babuschkin, X. Chen, P.-S. Huang, J. Welbl, S. Gowal, A. Cherepanov, J. Molloy, D. J. Mankowitz, E. S. Robson, P. Kohli, N. de Freitas, K. Kavukcuoglu, and O. Vinyals, *Competition-level code generation with alphacode*, Sci. **378** (2022), no. 6624, 1092–1097.
6. E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, *Codegen: An open large language model for code with multi-turn program synthesis*, (International Conference on Learning Representations), 2022. https://api.semanticscholar.org/CorpusID:252668917
7. K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, *BLEU: A method for automatic evaluation of machine translation*, (Association for Computational Linguistics, Philadelphia, PA, USA), 2002, pp. 311–318.
8. S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, *Codebleu: a method for automatic evaluation of code synthesis*, arXiv preprint, 2020, DOI 10.48550/arXiv.2009.10297
9. S. Kulal, P. Pasupat, K. Chandra, M. Lee, O. Padon, A. Aiken, and P. S. Liang, SPoC: Search-based Pseudocode to Code, Proceedings of the 33rd International Conference on Neural Information Processing Systems *Edited by* H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Vol. **32**, Curran Associates, Inc., 2019.
10. Y. Feng, S. Vanam, M. Cherukupally, W. Zheng, M. Qiu, and H. Chen, *Investigating code generation performance of chatgpt with crowdsourcing social data*, (IEEE 47th Annual Computers, Software, and Applications conference (COMPSAC), Torino, Italy), 2023, pp. 876–885.
11. N. Nguyen and S. Nadi, *An empirical evaluation of github copilot's code suggestions*, (IEEE/ACM 19th International Conference on Mining Software Repositories (MSR), Pittsburgh, PA, USA), 2022, pp. 1–5.
12. B. Yetistiren, I. Ozsoy, and E. Tuzun, *Assessing the quality of github copilot's code generation*, (Proc. of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering, Singapore), 2022, pp. 62–71.
13. T. Kim, Y. Jang, C. Lee, H. Koo, and H. Kim, *Smartmark: Software watermarking scheme for smart contracts*, (IEEE/ACM 45th International Conference on Software Engineering (ICSE), Melbourne, Australia), 2023, pp. 283–294.
14. J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, and C. Sutton, *Program synthesis with large language models*, arXiv preprint, 2021. DOI 10.48550/arXiv.2108.07732
15. Q. Zheng, X. Xia, X. Zou, Y. Dong, S. Wang, Y. Xue, Z. Wang, L. Shen, A. Wang, Y. Li, T. Su, Z. Yang, and J. Tang, *CodeGeeX: A pre-trained model for code generation with multilingual evaluations on humaneval-XX*, (Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Long Beach, CA, USA), 2023, pp. 5673–5684.
16. B. Athiwaratkun, S. K. Gouda, Z. Wang, X. Li, Y. Tian, M. Tan, W. U. Ahmad, S. Wang, Q. Sun, and M. Shang, *Multilingual evaluation of code generation models*, (The Eleventh International Conference on Learning Representations), 2023.
17. B. Roziére, 2023. Code llama: Open foundation models for code.
18. D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, A. Arora, E. Guo, C. Burns, S. Puranik, H. He, D. Song, and J. Steinhardt, *Measuring coding challenge competence with apps*, (Proc. of the Neural Information Processing Systems Track on Datasets and Benchmarks), 2021.
19. X.-Y. Li, J.-T. Xue, Z. Xie, and M. Li, *Think outside the code: Brainstorming boosts large language models in code generation*, arXiv preprint, 2023, DOI 10.48550/arXiv.2305.10679
20. OpenAI, Gpt-4 technical report, 2023.

AUTHOR BIOGRAPHIES

Sangyeop Yeo received the BS degree from Kumoh National Institute of Technology in 2023. He is currently a graduate student at the Artificial Intelligence Division of the University of Science and Technology, Republic of Korea. His research interests include code generation, natural language processing, security, and machine learning

Yu-Seung Ma received the BS, MS, and PhD degrees in computer science from the Korea Advanced Institute of Science and Technology, Republic of Korea, in 1998, 2000, and 2005, respectively. In 2005, she joined the Electronics and Telecommunications Research Institute, where she currently serves as a principal researcher. She is also a professor at the Artificial Intelligence Division, University of Science and Technology, Republic of Korea. Her research interests include code generation, software engineering, and machine learning.

Sang Cheol Kim received the BS degree in electronics and electrical engineering from Kyungbuk National University, Kyungbuk, Republic of Korea, in 1999. He received the MS and PhD degrees from the Pohang University of Science and Technology, Pohang, Republic of Korea, in 2001 and 2006, respectively. He has been working for the Electronics and Telecommunications Research Institute, Republic of Korea, since 2006. He has developed a variety of operating systems for real-time systems, including NanoQplus, an OS for sensor networks, ROSEK, an international standard automotive OS, and NEOS, a real-time OS for multicore real-time systems in military fields. His research interests include operating systems for artificial intelligence and text-to-code generation in large language models.

Hyungkook Jun received the BS degree in computer science and engineering from Sungkyunkwan University, Republic of Korea, in 1999, and the MS degree in electrical and computer engineering from Sungkyunkwan University in 2001. His major research interests include embedded systems, communication middleware, cyber-physical systems, multiaccess edge computing, neuromorphic systems, machine learning operations, and large language models. He is currently a principal member of the engineering staff at the Electronics and Telecommunications Research Institute, Republic of Korea.

Taeho Kim received the BS degree from Sungkyunkwan University, Republic of Korea, in 1995, and the MS and PhD degrees in computer science from the Korea Advanced Institute of Science and Technology, Republic of Korea, in 1997 and 2005, respectively. He is currently a principal researcher at the Artificial Intelligence Computing Research Lab, Electronics and Telecommunications Research Institute, Republic of Korea. His research interests include trustworthy software engineering based on artificial intelligence, deep learning compilers for artificial intelligence accelerators, security- and safety-critical cyber-physical systems, and real-time operating systems.

How to cite this article: S. Yeo, Y.-S. Ma, S. C. Kim, H. Jun, and T. Kim, *Framework for evaluating code generation ability of large language models*, ETRI Journal **46** (2024), 106–117, DOI 10.4218/etrij.2023-0357