НАУЧНО-ТЕХНИЧЕСКИЙ ВЕСТНИК ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ
январь–февраль 2026 Том 26 № 1 http://ntv.ifmo.ru/
SCIENTIFIC AND TECHNICAL JOURNAL OF INFORMATION TECHNOLOGIES, MECHANICS AND OPTICS
January–February 2026 Vol. 26 No 1 http://ntv.ifmo.ru/en/
ISSN 2226-1494 (print) ISSN 2500-0373 (online)

doi: 10.17586/2226-1494-2026-26-1-135-144

Natural language processing metrics efficiency for evaluating a generated code: facing the challenge

Dmitriy V. Fedrushkov[1], Sergey V. Kovalchuk[2]✉, Artem A. Aliev[3]

[1,2] ITMO University, Saint Petersburg, 197101, Russian Federation
[3] St. Petersburg State University (SPbSU), Saint Petersburg, 199034, Russian Federation

[1] dvfedrush@gmail.com, https://orcid.org/0009-0003-9088-4056
[2] kovalchuk@itmo.ru✉, https://orcid.org/0000-0001-8828-4615
[3] artem.aliev@gmail.com, https://orcid.org/0000-0001-7984-4721

Abstract
The evaluation of Large Language Models (LLMs) for code generation tasks presents unique challenges, because conventional Natural Language Processing (NLP) methods might be not applicable for assessing the code. Traditional text similarity metrics may fail to capture the functional correctness of generated code. This study investigates the effectiveness of various evaluation metrics by comparing a LLM-generated code with the mutated versions of the original code snippets. Using state-of-the-art models and benchmarks, the generated and the mutated codes were evaluated using some widely used NLP metrics, including code-oriented CodeBLEU and Ruby, and the neural network-based BERTScore and CodeBERTScore. Results demonstrated that text-oriented metrics tend to have inferior relevance in assessing programming tasks, particularly when functional accuracy is crucial. Code-specific and neural metrics show higher correlation with test pass rates, although their limitations highlight the need for a further refinement. The findings underscore the importance of developing functionality-aware evaluation methods for LLM-driven code generation. This research suggests insights into metrics selection to assess the quality of AI-generated code.

Keywords
large language models, natural language processing, code generation, metrics, evaluation, source code mutations

For citation: Fedrushkov D.V., Kovalchuk S.V., Aliev A.A. Natural language processing metrics efficiency for evaluating a generated code: facing the challenge. *Scientific and Technical Journal of Information Technologies, Mechanics and Optics*, 2026, vol. 26, no. 1, pp. 135–144. doi: 10.17586/2226-1494-2026-26-1-135-144

УДК 004.896

Сложности использования метрик для обработки естественного языка при оценке сгенерированного кода

Дмитрий Витальевич Федрушков[1], Сергей Валерьевич Ковальчук[2]✉, Артем Александрович Алиев[3]

[1,2] Университет ИТМО, Санкт-Петербург, 197101, Российская Федерация
[3] Санкт-Петербургский государственный университет, Санкт-Петербург, 199034, Российская Федерация

[1] dvfedrush@gmail.com, https://orcid.org/0009-0003-9088-4056
[2] kovalchuk@itmo.ru✉, https://orcid.org/0000-0001-8828-4615
[3] artem.aliev@gmail.com, https://orcid.org/0000-0001-7984-4721

Аннотация
Оценка больших языковых моделей (Large Language Models, LLM) для задач генерации кода представляет собой особую сложность, поскольку традиционные методы обработки естественного языка (Natural Language Processing, NLP) могут быть неприменимы для оценки кода. Классические метрики сходства текста могут не отражать функциональную корректность сгенерированного кода. В работе изучается эффективность различных метрик оценки путем сравнения кода, сгенерированного LLM, с мутированными версиями исходных фрагментов кода. Используя современные модели и бенчмарки, были оценены сгенерированные и мутированные фрагменты

кода с применением нескольких широко используемых метрик NLP, включая ориентированные на программный код CodeBLEU и Ruby, а также нейросетевые BERTScore и CodeBERTScore. Показано, что ориентированные на оценку текста метрики (а именно BiLingual Evaluation Understudy и Recall-Oriented Understudy for Gisting Evaluation) имеют тенденцию к снижению релевантности при оценке задач программирования, особенно когда функциональная точность критически важна. Ориентированные на код и нейросетевые метрики демонстрируют более высокую корреляцию с показателями прохождения тестов, хотя их ограничения указывают на необходимость дальнейшего совершенствования. Полученные результаты подтверждают важность разработки методов оценки, учитывающих функциональность, для генерации кода на основе LLM. Предложены новые подходы к выбору метрик для оценки качества кода, сгенерированного искусственным интеллектом.

Ключевые слова
большие языковые модели, обработка естественного языка, генерация кода, метрики, оценка качества, мутации исходного кода

Ссылка для цитирования: Федрушков Д.В., Ковальчук С.В., Алиев А.А. Сложности использования метрик для обработки естественного языка при оценке сгенерированного кода // Научно-технический вестник информационных технологий, механики и оптики. 2026. Т. 26, № 1. С. 135–144 (на англ. яз.). doi: 10.17586/2226-1494-2026-26-1-135-144

Introduction

The advent of Large Language Models (LLMs) has transformed software development, providing tools to streamline coding and clarify some complex principles in software engineering. However, LLMs currently lack critical thinking capabilities which can result in some errors and inefficiencies in code generation workflows. For example, a recent report highlighted that integrating GitHub Copilot increased bug rates by 41 % without a corresponding improvement in productivity metrics, suggesting that LLM assistance may not inherently enhance software development efficiency.

The core challenge lies in how LLMs interpret and generate a code. While these models handle a code similarly to a text, effective code generation demands a deeper analysis that includes the code structure, dependency mapping, and logical coherence. Ignoring these facets can lead to flawed outputs, where the generated code lacks functional robustness and readability.

In view of the above, this study emphasizes the crucial role of evaluation metrics in improving code generation quality. Properly chosen metrics would not only enable more accurate assessments of the generated code but also provide insights into areas needing refinement, thereby enhancing the overall utility of LLMs in software development. Here, some widely adopted code assessment metrics were evaluated and compared, analyzing their effectiveness in measuring the code quality and reliability.

Background

The application of LLMs to code generation represents an important milestone in their development, extending their application beyond Natural Language Processing (NLP) to areas requiring a sophisticated understanding of structured languages. These models demonstrated potential in automating software development tasks like code synthesis, completion, and debugging. However, the integration of LLMs into software development pipelines introduces unique challenges, particularly in assessing their outputs. As presented by recent studies [1, 2], existing evaluation metrics might fail to capture the numerous dimensions of code quality and functionality. Therefore, the usage of these metrics can be unrepresentative and it needs further analysis of their efficiency in terms of estimating the quality of generative Artificial Intelligence (AI) output.

Metrics originally developed for NLP often struggle to capture the subtleties of software code. For example, although a generated code fragment may exhibit high lexical similarity to a reference solution, it may still malfunction due to logical errors or violations of program requirements. This gap highlights the need for specialized evaluation metrics that can assess both syntactic accuracy and functional correctness.

To address these issues, researchers introduced a number of metrics, particularly CodeBLEU [3] and Ruby [4], which incorporate code-specific features into their assessments. Standardized benchmarks like HumanEval [5, 6] and CoderEval [7] provide a basis for evaluating the ability of LLMs to generate functionally correct code. Nevertheless, these advancements remain a work in progress, and there is no consensus on the most effective strategies for evaluating AI-driven code generation.

This study examines existing metrics and benchmarks and identifies areas for further improvement. By exploring the limitations of current approaches, it aims to provide a clearer understanding of the challenges associated with evaluating LLM in code generation and suggest potential directions for future research in this evolving field.

Methods

Suggested approach focuses on assessing the effectiveness of various evaluation metrics in estimating the quality of code generated by LLMs. Using well-known metrics in combination with a measure of functional correctness, namely Pass@1, the evaluation results of the code generated by LLM and the evaluation of the code with minor modifications were compared, where modifications were obtained using a mutation testing tool. Mutation testing, commonly used to evaluate software test suites, introduces minor alterations in the source code to simulate potential errors and gauge the response of evaluation metrics. Since mutation testing typically results in minimal source code changes, these modifications are expected to have limited impact on evaluation metrics like readability and structural integrity but to significantly

affect metrics related to functional correctness, particularly passing probability.

Generated code vs. mutated code

To modify the source code, small mutations were injected to original code snippets, including altering operators, modifying variable assignments, and adjusting conditional statements. These mutations simulate common programming errors without impacting code readability or structural coherence. It guarantees that changes of metrics scores are directly attributable to functional correctness rather than extensive structural transformations.

Code solutions were generated for each prompt using a state-of-the-art LLM, which were then compared to the mutated versions of the original code snippets. To conduct the comparison of given code snippets, both the LLM-generated code and the mutated originals were evaluated using a range of metrics assessing text similarity, structural integrity, and text embeddings distance. Notably, the passing probability metric was used to estimate the likelihood of each code variant passing a standard test suite, hypothesizing a significant decrease in the passing probability for the mutated code due to the errors introduced.

Metric Comparison and Impact Analysis

The study suggests the examination of the extent to which the scores provided by each NLP metric aligns with functional outcomes, focusing on situations where evaluating results may conflict. The point of view in this research is that significant discrepancies between the passing tests and NLP metrics scores might indicate the insufficiency of conventional NLP metrics when evaluating code fragments. For instance, a high similarity score does not always guarantee that the code would pass functional tests, while low scores may still correspond to a functionally correct code. By identifying these conflicts, the limitations of each metric in accurately representing code quality were assessed, especially in the context of comparing the LLM-generated code vs. minimally modified (mutated) code.

Additionally, the study analyzed the capabilities of both language models to follow the alternative generation way for each mutated code. Results include the compared probabilities of canonical and altered next tokens at the places in code, where the mutations were injected. This helps to support or disprove the view of NLP metrics when evaluating generated code.

Experimental setup

The experimental setup included evaluation of both LLM-generated code and mutated code using several benchmarks, models, and metrics to systematically evaluate the quality and functionality of the generated code fragments. The selected benchmarks, models, and metrics were chosen to provide a comprehensive view of how different assessing approaches capture code quality and functional correctness, especially when comparing generated code with mutated versions of the original code fragments.

Benchmarks

— HumanEval-X [5, 6]: a widely used benchmark in code generation tasks that evaluates the functional correctness of generated code. It consists of a set of programming tasks with well-defined input-output specifications, which allows for robust testing of functional accuracy.
— CoderEval [7]: a benchmark that provides a diverse set of programming tasks with varying levels of complexity, making it suitable for evaluating both simple and complex code generation capabilities. It is designed to evaluate code generation models from various aspects of software development.

Models

— Codegen-2B-Multi [8]: a 2 billion parameter model that was designed specifically for code generation tasks with the usage of multiple programming languages. This provides a possibility to be used for both general-purpose programming and programming language-dependent tasks, which makes it versatile for generating code in different contexts.
— MetaLlama-3-8B [9, 10]: an 8 billion parameter language model designed for complex natural language tasks. Even though this model was trained to handle the natural language tasks, the large parameter size provides improved generation capabilities, which allows it to handle complex and subtle programming tasks.

Mutation testing

Proposed analysis methods used the Universalmutator [11, 12] tool to apply systematic mutations to the original code fragments. Universalmutator makes minor changes, including operator substitutions, conditional negations, and variable modifications that simulate common coding errors while preserving the overall structure of the code. This helped to compare the code generated by LLM with the mutated code fragments, highlighting the sensitivity of each evaluation metric to small code changes and its ability to capture functional inconsistencies.

Metrics

This study employed a diverse set of evaluation metrics to measure the differences between canonical solutions and candidate code snippets. Each code fragment obtained from language models and a mutation testing tool was evaluated by all these metrics. Selected NLP metrics might provide insights into various aspects of code assessment, including exact textual matches, structural alignment, and semantic similarity.

— Exact Match (EM): It measures whether the generated text matches the expected text exactly. This metric is strict and is mainly used to evaluate simple code snippets.
— Edit Similarity (ES): This metric calculates the minimal number of edit operations (insertions, deletions, substitutions) needed to transform generated text into reference text. It captures textual similarity, with lower values indicating closer matches.
— BiLingual Evaluation Understudy (BLEU) [13]: BLEU score was designed for NLP challenges, it evaluates *n*-gram overlap between generated and reference text. In the context of code-related tasks, BLEU is useful for measuring basic structural alignment in code.
— Recall-Oriented Understudy for Gisting Evaluation (ROUGE) [14]: ROUGE is another metric designed for NLP tasks. It is based on recall of *n*-grams and is used here to measure similarity in terms of code structure and language patterns.

— CodeBLEU [3]: CodeBLEU extends BLEU with syntax-based and AST-based components; it was specifically designed for assessing the code. This metric takes into account structural and syntactic aspects of the code.
— Ruby [4]: Ruby is a metric designed for assessing code functionality and readability. It analyzes both surface-level patterns and deeper functional aspects, providing a more holistic evaluation of generated code.
— BERTScore [15]: BERTScore evaluates semantic similarity between generated and reference code snippets, using pre-trained BERT embeddings and focusing on the contextual similarity rather than exact matches.
— CodeBERTScore [16]: A modification of BERTScore with use of fine-tuned BERT model for code tasks. CodeBERTScore uses the embeddings given from CodeBERT model to assess similarity in code generation tasks.

With the use of the Codegen-2B-Multi and MetaLlama-3-8B models, code solutions were generated for each programming task provided in benchmarks. Universalmutator tool was applied to the original code snippets to create a set of mutated versions for comparison. Each LLM-generated fragment and mutated original were evaluated using a set of selected NLP metrics to capture differences in assessing code quality, functionality, and readability.

This setup helped to assess the effectiveness of each metric in evaluating LLM-generated code, particularly in scenarios where generated or mutated code failed functionality tests but got high scores from NLP metrics or vice versa. Suggested experimental framework provided a comprehensive assessment of each metric performance in different code generation contexts, highlighting their limitations in use for practical LLM code generation.

The whole pipeline is shown in the Figure.

Results

Recent studies [1, 2] revealed the limitations of evaluating the code generated by LLMs, particularly, insufficient test-based evaluation and a restricted scope of evaluation. However, they do not seem to suggest any analysis of the above limitations and, namely, of the correlation between the different metrics and the code functionality was not proposed. This research suggested selecting a set of metrics and providing their in-depth analysis to assess the quality of LLM generated code. This analysis appears to have revealed relevant metrics.

Table 1 presents the evaluation results for the Codegen 2B Multi model across multiple programming languages and metrics. The table includes both text-oriented metrics and code-specific metrics. It shows the model performance in terms of its ability to produce code snippets that match human-written code, as well as its functional correctness, as measured by Pass@1.

Table 2 demonstrates the evaluation results for the MetaLlama-3-8B model using the same metrics. This provides a direct comparison of MetaLlama's and Codegen's performance and an insight into how larger and more advanced models handle code generation tasks.

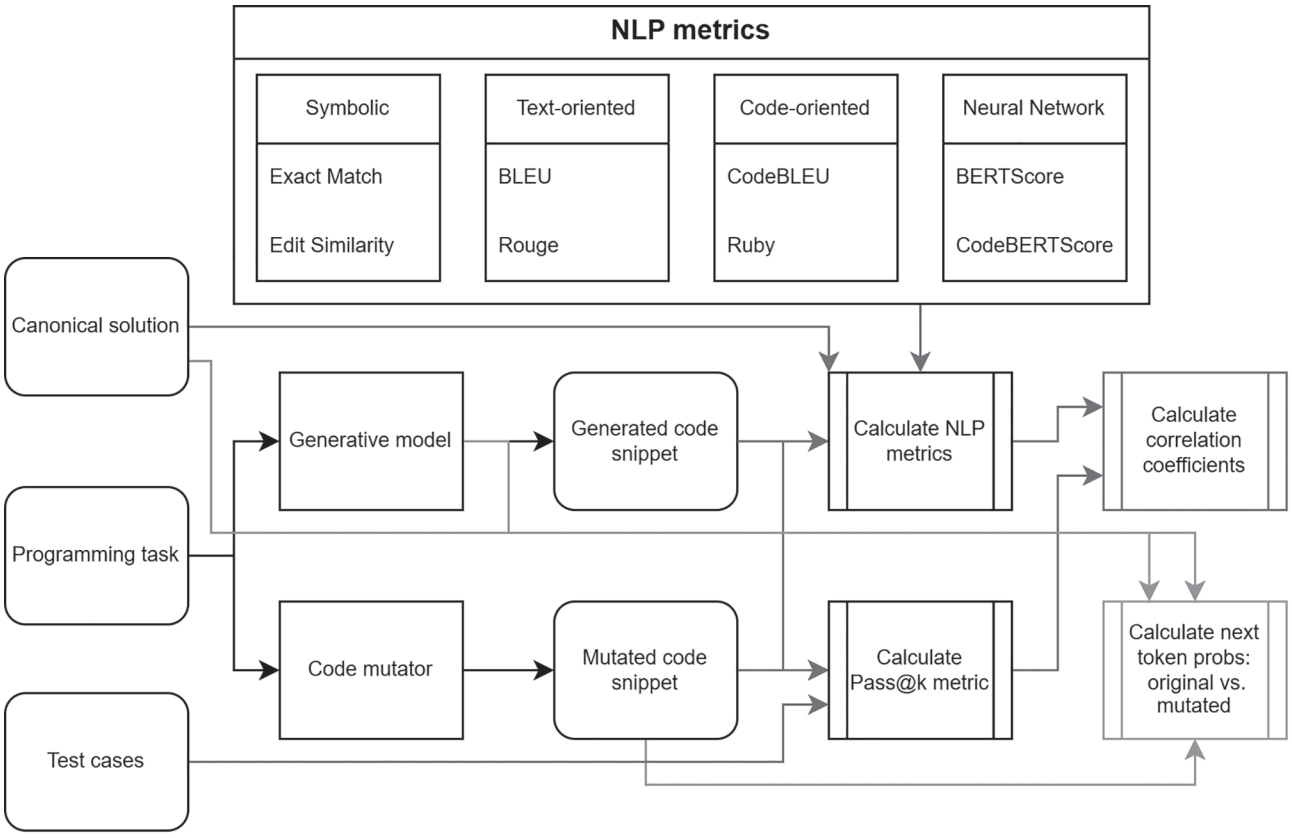

Figure. Metric collection pipeline

Table 1. Codegen-2B-Multi evaluation results

Metric	HumanEval				CoderEval	
	Python	Java	Go	JS	Python	Java
EM	0.000	0.000	0.000	0.000	0.000	0.000
ES	0.203	0.424	0.276	0.269	0.165	0.168
BLEU	0.066	0.258	0.157	0.129	0.038	0.075
ROUGE	0.243	0.439	0.343	0.319	0.177	0.216
CodeBLEU	0.239	0.328	0.324	0.250	0.206	0.251
Ruby	0.136	0.335	0.221	0.212	0.120	0.134
BERTScore	0.785	0.880	0.823	0.828	0.782	0.797
CodeBERTScore	0.710	0.790	0.799	0.756	0.677	0.708
Pass@1	0.140	0.134	0.055	0.073	0.057	0.013

Table 2. MetaLlama evaluation results

Metric	HumanEval				CoderEval	
	Python	Java	Go	JS	Python	Java
EM	0.012	0.000	0.000	0.000	0.000	0.000
ES	0.379	0.471	0.342	0.311	0.173	0.221
BLEU	0.182	0.323	0.205	0.179	0.046	0.127
ROUGE	0.447	0.494	0.412	0.379	0.186	0.279
CodeBLEU	0.282	0.367	0.359	0.288	0.197	0.262
Ruby	0.301	0.396	0.276	0.257	0.117	0.198
BERTScore	0.844	0.890	0.847	0.846	0.777	0.823
CodeBERTScore	0.795	0.810	0.819	0.784	0.676	0.737
Pass@1	0.348	0.348	0.226	0.213	0.157	0.065

Particularly noticeable in this table is the discrepancy between textual and code-specific metrics as well as their correlation with functional correctness.

Table 3 plots the evaluation results for the mutated code snippets using the selected set of metrics. It highlights the conflict between NLP metrics and measures of passing probability. While the number of NLP metrics returns the extremely high scores, the frequency of passing test cases does not overcome the results of code given from generative models. Based on the analysis of the results and considering the Pass@1 metric as the most relevant to human judgment, it appears to become evident that no single text similarity metric adequately captures the correctness of answers generated by LLMs.

Text-oriented metrics, namely BLEU, ROUGE, ES and EM, seem to fail to effectively assess functional correctness in code generation tasks. Other metrics, including neural network-based scores (BERT and CodeBERT) and

Table 3. Mutated code evaluation results

Metric	HumanEval				CoderEval	
	Python	Java	Go	JS	Python	Java
EM	0.000	0.018	0.000	0.000	0.026	0.035
ES	0.923	0.960	0.947	0.942	0.909	0.937
BLEU	0.818	0.907	0.895	0.896	0.881	0.893
ROUGE	0.916	0.963	0.963	0.962	0.932	0.960
CodeBLEU	0.825	0.908	0.904	0.874	0.860	0.869
Ruby	0.878	0.943	0.935	0.929	0.902	0.917
BERTScore	0.973	0.990	0.988	0.988	0.982	0.986
CodeBERTScore	0.963	0.981	0.983	0.978	0.971	0.979
Pass@1	0.037	0.122	0.079	0.091	0.039	0.121

code-oriented (CodeBLEU and Ruby) also showed their ineffectiveness when trying to evaluate code in terms of its functional correctness. Moreover, evaluation results represented not only the inefficiency of NLP metrics in the scope of assessing code functionality, but also the discrepancies with the test-based evaluation method. The CoderEval results presented in Table 3 appear to confirm the aforementioned — metrics designed to evaluate the language models in common natural language tasks might be not applicable for assessing the issues related to software development.

The insufficiency of text-based metrics is especially obvious in the results of the assessing mutated code by these metrics. Despite the high scores of all the metrics, which used to compare the candidate code snippet with the original one, the passing probability of the corresponding code fragments. To support this evidence, an analysis of NLP metrics correlation with the passing probability was conducted. Table 4 represents the correlation coefficients of each NLP metric.

Finally, suggested analysis examined the tokens probabilities distribution of chosen models at the place in code, where the source code was mutated. The probability of models to sample "right" (original) and "mutated" tokens was assessed. Table 5 plots the results of these experiments across all the chosen models, benchmarks, and programming languages.

All these might indicate that the NLP metrics should not be used alone to assess the code generation models. The findings above highlight the need for further refinement of evaluation metrics to ensure they can accurately capture both the functional correctness and the structural quality in generated code.

Evaluation example

The several randomly chosen tests in the CoderEval experimental evaluation were inspected and it is shown how the NLP metrics measured the similarity scores. For example, for the following task (CoderEval 636767781a6d9265ec018242) the following query provided:

```
/**Call the doAppend method on all
attached appenders. **/
public int appendLoopOnAppenders(Loggi
ngEvent event){
```

Ground truth solution given by benchmark presented as follows:

```
  int size=0;
  Appender appender;
  if (appenderList != null) {
    size=appenderList.size();
    for (int i=0; i < size; i++) {
      appender=(Appender)appenderList.
elementAt(i);
      appender.doAppend(event);
    }
  }
  return size;
}
```

Codegen language model generated the next solution:

```
  for(int i=0;i<this._loggers.
size();i++){
    this._loggers.get(i).
appendLoopOnAppenders(event);
  }

  for(int i=0;i<this._appenders.
size();i++){
    this._appenders.get(i).
appendLoopOnAppenders(event);
  }
}
public void appendLogIfAny(LoggingEvent
event){
  for(int i=0;i<this._loggers.
size();i++){
    this._loggers.get(i).
appendLogIfAny(event);
  }
  for(int i=0;i<this._appenders.
size();i++){
    this._appenders.get(i).
appendLogIfAny(event);
  }
}
/**
```

Table 4. Correlation coefficients

Metric	HumanEval				CoderEval		Overall
	Python	Java	Go	JS	Python	Java	
EM	0.055	0.026	0.000	0.000	–0.016	0.000	0.023
ES	–0.139	–0.022	–0.066	–0.078	–0.016	0.141	–0.013
BLEU	–0.169	–0.027	–0.065	–0.066	–0.005	0.149	–0.027
ROUGE	–0.117	–0.027	–0.055	–0.056	0.002	0.140	–0.005
CodeBLEU	–0.104	–0.044	–0.049	–0.055	–0.002	0.142	–0.017
Ruby	–0.127	–0.008	–0.059	–0.060	–0.062	0.151	–0.011
BERTScore	–0.136	–0.009	–0.044	–0.064	–0.093	0.133	–0.014
CodeBERTScore	–0.093	–0.004	–0.029	–0.052	–0.061	0.134	0.004

Table 5. Probabilities of next tokens, %

Next token	Model	Benchmark	Language	Temperature (arbitrary unit)			
				0.2000	0.6000	0.8000	1.0000
“Mutated”	Codegen	HumanEval	Python	0.5841	0.4464	0.4041	0.3678
			Java	0.0000	0.0000	0.0002	0.0015
			Go	0.6971	0.8319	0.8383	0.8174
			JavaScript	2.2167	2.3026	2.2212	2.0769
			Overall	0.8745	0.8952	0.8659	0.8159
		CoderEval	Python	0.8696	0.8491	0.8248	0.8092
			Java	0.4820	0.7722	0.8737	0.9453
			Overall	0.6758	0.8106	0.8493	0.8772
	Llama	HumanEval	Python	0.5582	0.3480	0.3000	0.2584
			Java	0.0000	0.0004	0.0041	0.0168
			Go	0.0000	0.0026	0.0117	0.0288
			JavaScript	0.0000	0.0070	0.0290	0.0660
			Overall	0.1396	0.0895	0.0862	0.0925
		CoderEval	Python	0.0000	0.0015	0.0100	0.0327
			Java	0.0147	0.0579	0.0653	0.0814
			Overall	0.0073	0.0297	0.0376	0.0571
“Right”	Codegen	HumanEval	Python	0.0257	0.1575	0.1849	0.1966
			Java	0.0000	0.0000	0.0001	0.0008
			Go	0.0000	0.0025	0.0148	0.0442
			JavaScript	0.0000	0.0052	0.0229	0.0545
			Overall	0.0064	0.0413	0.0557	0.0740
		CoderEval	Python	0.0000	0.0119	0.0350	0.0725
			Java	8.7885	8.4047	8.2106	7.9311
			Overall	4.3943	4.2083	4.1228	4.0018
	Llama	HumanEval	Python	0.0357	0.1408	0.1530	0.1588
			Java	0.0000	0.0014	0.0147	0.0594
			Go	0.0000	0.0004	0.0037	0.0146
			JavaScript	0.0000	0.0041	0.0202	0.0538
			Overall	0.0089	0.0367	0.0479	0.0716
		CoderEval	Python	0.0000	0.0021	0.0118	0.0365
			Java	0.1915	0.2376	0.2676	0.2943
			Overall	0.0958	0.1198	0.1397	0.1654

```
* This method will create a LogAdapter
or will return null in case of
failure.
* @return A LogAdapter or null.
*/
public LogAdapter createLogAdapter(){
  return createLogAdapter(false);
}
/**
* This method will create a LogAdapter
or will return null in
```

Llama language model suggested the following:

```
    int i = 0;
    int result = 0;
    //if (event.getThreadPriority()!=
null)
    //    event.
setThreadPriority(event.
getThreadPriority().intValue());

    // Set thread name
    if (event.getThreadName()!= null)
        event.setThreadName(event.
getThreadName().toString());
```

```
     // Set Hostname
    if (event.getThreadName()!= null)
        event.setHostName(event.
getThreadName().toString());

    // Set hostname
    if (event.getThreadName()!= null)
        event.setHostName(event.
getThreadName().toString());

    // Set application name
    if (event.getApplicationName()!=
null)
       event.setApplicationName(event.
getApplicationName().toString());

    // Set log file
    if (event.getLogFile()!= null)
        event.setLogFile(event.
getLogFile().toString());

    // Set logger
    if (event.getLoggerName()!= null)
        event.setLoggerName(event.
getLoggerName().toString());

    // Set log level
    if (event.getLevel()!= null)
        event.setLevel(event.
getLevel().toString());

    // Set thread priority
    if (event
```

Minorly mutated code presented as follows:

```
    int size = 0;
    Appender appender;

    if(appenderList != null) {
      size = appenderList.size();
      for(int i = 0; i < size; i++) {
appender = (Appender) appenderList.
elementAt(i);
appender.doAppend(event);
      }
    }
    return size;
    break;
}
```

Evaluation results measured by selected NLP metrics and running the test cases represented as shown in Table 6. Finally, the next token probabilities determined as follows: Codegen mutated token probability equals 0.009058, Codegen canonical token probability is 0.01318, Llama mutated token probability amounts to 0.001244, Llama canonical token probability has a value of 0.005753.

Discussion

The main goal of this research was to find the most relevant metrics that can estimate the code generated by language models without using large computing resources to carry out this evaluation. This is important to enable the use of artificial intelligence methods in software development using developer devices. Since creating test coverage for each piece of code takes a lot of time, and testing itself uses a lot of calculations, it is not always possible to use test passability metrics to assess the generated code quality which justifies the need to use alternative methods for assessing the results obtained.

Importantly, this study did not involve test-based assessment methods using tests generated by language models. Although this method is quite promising, its use today is severely limited by the insufficient capabilities of language models in creating test coverage. Also, the use of such methods requires even more computing power, which defeats the purpose of this study.

Additionally, the results of code generation were not estimated from the perspective of various methods of evaluating code efficiency, namely execution time, memory usage and checking for vulnerabilities. These factors are known as quite important in assessing the code; however, its implementation also requires much computational resources that did not match with the goals of research.

Since the goal of finding better evaluation methods is to improve the use of artificial intelligence in software development problems, future work will focus on creating a developer-friendly environment in which these methods are not only useful in development, but also do not consume the majority of the developer's computing resources. This seems to allow faster software development, since it will take into account not only the developer's limited resources, but also the context of the project in which the work is being carried out.

A key limitation of this study lies in the relatively narrow scope of the samples, datasets, models, and evaluation tools utilized. Given the constraints of small computational resources, expanding these elements in future work presents an important research direction. Another notable limitation is the limited depth of

Table 6. CoderEval evaluation results example

Model	EM	ES	BLEU	Rouge	Code BLEU	Ruby	BERT Score	CodeBERT Score	Passed
CodeGen	0	0.2323	0.1145	0.2481	0.2786	0.1883	0.8328	0.7214	False
Llama	0	0.1522	0.0387	0.1212	0.1936	0.1009	0.7831	0.6667	False
UniversalMutator	0	0.8192	1.0000	1.0000	0.6765	1.0000	1.0000	0.9809	False

analysis. While the results offer valuable insights, a more detailed exploration of the alignment and, critically, the discrepancies between metric-based and test-based evaluation is necessary. Furthermore, the quantity of data that the open testing benchmarks contain is not large enough to allow comprehensive analysis and clear conclusions to be drawn.

Conclusion

This study was conducted to examine the contribution of various metrics in assessing the relevance and performance of code generated by language models. The obtained results appear to indicate that the evaluation metrics that exist today are not capable of comprehensively assessing program code. However, using a combination of some of the existing methods can reflect the suitability of the code generated by artificial intelligence for the task at hand. The integration of adaptive evaluation pipelines may help capture nuances in code behavior that static metrics often overlook. Such advancements could ultimately lead to more reliable and context-sensitive approaches for assessing the functional quality of generated programs. Further research would involve Reinforcement Learning from Artificial Intelligence Feedback to conduct deeper analysis of generated code.

References

1. Chen L., Guo Q., Jia H., Zeng Z., Wang X., Xu Y., et al. A survey on evaluating large language models in code generation tasks. *arXiv*, 2025. arXiv:2408.16498v2. https://doi.org/10.48550/arXiv.2408.16498
2. Wang J., Chen Y. A review on code generation with LLMs: application and evaluation. *Proc. of the IEEE International Conference on Medical Artificial Intelligence (MedAI)*, 2023, pp. 284–289. https://doi.org/10.1109/medai59581.2023.00044
3. Ren S., Guo D., Lu S., Zhou L., Liu S., Tang D., et al. CodeBLEU: a method for automatic evaluation of code synthesis. *arXiv*, 2020. arXiv:2009.10297v2. https://doi.org/10.48550/arXiv.2009.10297
4. Tran N, Tran H., Nguyen S., Nguyen H., Nguyen T.N. Does BLEU score work for code migration? *Proc. of the IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, 2019, pp. 165–176. https://doi.org/10.1109/ICPC.2019.00034
5. Chen M., Tworek J., Jun H., Yuan Q., de Oliveira Pinto H.P., Kaplan J., et al. Evaluating large language models trained on code. *arXiv*, 2021. arXiv:2107.03374v2. https://doi.org/10.48550/arXiv.2107.03374
6. Zheng Q., Xia X., Zou X., Dong Y., Wang S., Xue Y., et al. CodeGeeX: a pre-trained model for code generation with multilingual benchmarking on HumanEval-X. *arXiv*, 2024. arXiv:2303.17568v2. https://doi.org/10.48550/arXiv.2303.17568
7. Yu H., Shen B., Ran D., Zhang J., Zhang Q., Ma Y., et al. CoderEval: a benchmark of pragmatic code generation with generative pre-trained models. *Proc. of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–12. https://doi.org/10.1145/3597503.3623316
8. Nijkamp E., Pang B., Hayashi H., Tu L., Wang H., Zhou Y., et al. CodeGen: an open large language model for code with multi-turn program synthesis. *arXiv*, 2023. arXiv:2203.13474v5. https://doi.org/10.48550/arXiv.2203.13474
9. Touvron H., Lavril T., Izacard G., Martinet X., Lachaux M-A, Lacroix T., et al. LLaMA: open and efficient foundation language models. *arXiv*, 2023. arXiv:2302.13971v1. https://doi.org/10.48550/arXiv.2302.13971
10. Grattafiori A., Dubey A., Jauhri A., Pandey A., Kadian A., Al-Dahle A., et al. The Llama 3 herd of models. *arXiv*, 2024. arXiv:2407.21783v3. https://doi.org/10.48550/arXiv.2407.21783
11. Deb S., Jain K., van Tonder R., Le Goues C., Groce A. Syntax is all you need: a universal-language approach to mutant generation. *Proc. of the ACM on Software Engineering*, 2024, vol. 1, no. FSE, pp. 654–674. https://doi.org/10.1145/3643756
12. Groce A., Holmes J., Marinov D., Zhang L. An extensible, regular-expression-based tool for multi-language mutant generation. *Proc. of the 40th International Conference on Software Engineering: Companion Proceeedings*, 2018, pp. 25–284. https://doi.org/10.1145/3183440.3183485
13. Papineni K., Roukos S., Ward T., Zhu W.-J. BLEU: a method for automatic evaluation of machine translation. *Proc. of the 40th Annual Meeting on Association for Computational Linguistics*, 2002, pp. 311–318. https://doi.org/10.3115/1073083.1073135
14. Lin C.-Y. ROUGE: a package for automatic evaluation of summaries. *Text Summarization Branches Out*, 2004, pp. 74–81.

Литература

1. Chen L., Guo Q., Jia H., Zeng Z., Wang X., Xu Y., et al. A survey on evaluating large language models in code generation tasks // arXiv. 2025. arXiv:2408.16498v2. https://doi.org/10.48550/arXiv.2408.16498
2. Wang J., Chen Y. A review on code generation with LLMs: application and evaluation // Proc. of the IEEE International Conference on Medical Artificial Intelligence (MedAI). 2023. P. 284–289. https://doi.org/10.1109/medai59581.2023.00044
3. Ren S., Guo D., Lu S., Zhou L., Liu S., Tang D., et al. CodeBLEU: a method for automatic evaluation of code synthesis // arXiv. 2020. arXiv:2009.10297v2. https://doi.org/10.48550/arXiv.2009.10297
4. Tran N, Tran H., Nguyen S., Nguyen H., Nguyen T.N. Does BLEU score work for code migration? // Proc. of the IEEE/ACM 27th International Conference on Program Comprehension (ICPC). 2019. P. 165–176. https://doi.org/10.1109/ICPC.2019.00034
5. Chen M., Tworek J., Jun H., Yuan Q., de Oliveira Pinto H.P., Kaplan J., et al. Evaluating large language models trained on code // arXiv. 2021. arXiv:2107.03374v2. https://doi.org/10.48550/arXiv.2107.03374
6. Zheng Q., Xia X., Zou X., Dong Y., Wang S., Xue Y., et al. CodeGeeX: a pre-trained model for code generation with multilingual benchmarking on HumanEval-X // arXiv. 2024. arXiv:2303.17568v2. https://doi.org/10.48550/arXiv.2303.17568
7. Yu H., Shen B., Ran D., Zhang J., Zhang Q., Ma Y., et al. CoderEval: a benchmark of pragmatic code generation with generative pre-trained models // Proc. of the IEEE/ACM 46th International Conference on Software Engineering. 2024. P. 1–12. https://doi.org/10.1145/3597503.3623316
8. Nijkamp E., Pang B., Hayashi H., Tu L., Wang H., Zhou Y., et al. CodeGen: an open large language model for code with multi-turn program synthesis // arXiv. 2023. arXiv:2203.13474v5. https://doi.org/10.48550/arXiv.2203.13474
9. Touvron H., Lavril T., Izacard G., Martinet X., Lachaux M-A, Lacroix T., et al. LLaMA: open and efficient foundation language models // arXiv. 2023. arXiv:2302.13971v1. https://doi.org/10.48550/arXiv.2302.13971
10. Grattafiori A., Dubey A., Jauhri A., Pandey A., Kadian A., Al-Dahle A., et al. The Llama 3 herd of models // arXiv. 2024. arXiv:2407.21783v3. https://doi.org/10.48550/arXiv.2407.21783
11. Deb S., Jain K., van Tonder R., Le Goues C., Groce A. Syntax is all you need: a universal-language approach to mutant generation // Proc. of the ACM on Software Engineering. 2024. V. 1. N FSE. P. 654–674. https://doi.org/10.1145/3643756
12. Groce A., Holmes J., Marinov D., Zhang L. An extensible, regular-expression-based tool for multi-language mutant generation // Proc. of the 40th International Conference on Software Engineering: Companion Proceeedings. 2018. P. 25–284. https://doi.org/10.1145/3183440.3183485
13. Papineni K., Roukos S., Ward T., Zhu W.-J. BLEU: a method for automatic evaluation of machine translation // Proc. of the 40th Annual Meeting on Association for Computational Linguistics. 2002. P. 311–318. https://doi.org/10.3115/1073083.1073135
14. Lin C.-Y. ROUGE: a package for automatic evaluation of summaries // Text Summarization Branches Out. 2004. P. 74–81.

15. Zhang T., Kishore V., Wu F., Weinberger K.Q., Artzi Y. BERTScore: evaluating text generation with BERT. *arXiv*, 2020. arXiv:1904.09675v3. https://doi.org/10.48550/arXiv.1904.09675
16. Zhou S., Alon U., Agarwal S., Neubig G. CodeBERTScore: evaluating code generation with pretrained models of code. *Proc. of the Conference on Empirical Methods in Natural Language Processing Proceedings*, 2023, pp. 13921–13937. https://doi.org/10.18653/v1/2023.emnlp-main.859

15. Zhang T., Kishore V., Wu F., Weinberger K.Q., Artzi Y. BERTScore: evaluating text generation with BERT // arXiv. 2020. arXiv:1904.09675v3. https://doi.org/10.48550/arXiv.1904.09675
16. Zhou S., Alon U., Agarwal S., Neubig G. CodeBERTScore: evaluating code generation with pretrained models of code // Proc. of the Conference on Empirical Methods in Natural Language Processing Proceedings. 2023. P. 13921–13937. https://doi.org/10.18653/v1/2023.emnlp-main.859

Authors

Dmitriy V. Fedrushkov — PhD Student, ITMO University, Saint Petersburg, 197101, Russian Federation, sc 57204596230, https://orcid.org/0009-0003-9088-4056, dvfedrush@gmail.com
Sergey V. Kovalchuk — PhD, Associate Professor, ITMO University, Saint Petersburg, 197101, Russian Federation, sc 55382199400, https://orcid.org/0000-0001-8828-4615, kovalchuk@itmo.ru

Artem A. Aliev — Senior Lecturer, St. Petersburg State University (SPbSU), Saint Petersburg, 199034, Russian Federation, sc 58188719500, https://orcid.org/0000-0001-7984-4721, artem.aliev@gmail.com

Авторы

Федрушков Дмитрий Витальевич — аспирант, Университет ИТМО, Санкт-Петербург, 197101, Российская Федерация, sc 57204596230, https://orcid.org/0009-0003-9088-4056, dvfedrush@gmail.com
Ковальчук Сергей Валерьевич — кандидат технических наук, доцент, Университет ИТМО, Санкт-Петербург, 197101, Российская Федерация, sc 55382199400, https://orcid.org/0000-0001-8828-4615, kovalchuk@itmo.ru
Алиев Артем Александрович — старший преподаватель, Санкт-Петербургский государственный университет, Санкт-Петербург, 199034, Российская Федерация, sc 58188719500, https://orcid.org/0000-0001-7984-4721, artem.aliev@gmail.com

Received 19.09.2025
Approved after reviewing 02.12.2025
Accepted 20.01.2026

Статья поступила в редакцию 19.09.2025
Одобрена после рецензирования 02.12.2025
Принята к печати 20.01.2026