ADAP: An Improved Evaluation Algorithm for Industrial Control Based on Large Language Models

Jiatong Zheng[1,2], Dong Li[1], Yong Sun[1], Chunhe Song[1]
[1]State KeyLaboratory of Robotics, Shenyang Institute of Automation, Chinese Academy of Sciences,Shenyang 110016, China
[2]University of Chinese Academy of Sciences, Beijing 100049, China
Corresponding authors: Dong Li and Chunhe Song (lidong@sia.cn,songchunhe@sia.cn)

Abstract

At present, large language models can help solve problems in fields with strong versatility. However, in some industrial control fields, such as the programmable logic controller (PLC) field, the versatility of large language models cannot solve the problem of compiling platform code application well. The general code generation problem can be solved, but the code generation problem in the industrial control field is difficult to use because different software platforms in the industrial control field have various commands that is hard to apply. In addition, code evaluations like BLEU and Pass@k have their shortcomings when they are used to evaluate generated codes because they have to fix the n in BLEU's n-gram and the k in Pass@k's k. In response to the above problems, this thesis firstly constructs the IC-100 industrial dataset to solve the problem of extremely scarce datasets in the industrial control field. Secondly, the dataset is tested by general large language models such as GPT-4o to generate code with code score evaluation. Third, this thesis proposes an algorithm called ADAP, which means adaptive n in BLEU's n-gram and k in Pass@k. BLEU and Pass@k with their ADAP algorithms are contrastingly used to examine whether the ADAP algorithms have better evaluation performance and code score evaluation algorithm is conducted to judge whether the language model generate a good industrial control code. Finally, the experimental results show that the ADAP algorithm has good code feedback when the feedback of the code score evaluation algorithm is feasible.The ADAP algorithm has improved 14% averagely compared with original algorithms in the IC-100 dataset. The IC-100 dataset and ADAP algorithm in this thesis can be well applied to code generation in the field of industrial control, bringing new inspiration to relevant practitioners and improving work efficiency.

Index Terms—Generative AI, Large Language Models, Industrial Control, IEC 61131-3, Structured Text

1 Introduction

In the past few years, generative AI has developed rapidly, showing its potential in many fields. Among them, code generation, as one of the important application areas of generative AI, has received more and more attention. Code generation technology aims to reduce the workload of developers and improve programming efficiency by automatically generating source code that meets specific requirements. In recent years, concepts such as Attention, Transformer, and NLP have gradually emerged, and popular large language models such as GPT, Llama, BERT, and Grok have further emerged.

Although LLMs have achieved remarkable results in code generation for general programming languages, their application in control logic engineering still has not solved the problem of high versatility and universality. In the industrial control field rules of IEC 61131-3, structured text is the most widely used programming language in the field of industrial Internet. Existing public LLMs are not trained specifically for control logic programming, so it is difficult for them to generate valid source code that complies with the IEC 61131-3 ST standard. Prompt words are an important part of this experiment and are the basis for the experiment of this paper. At present, the problems faced in the field of industrial control are: 1. There is no suitable high-quality prompt set; 2. There is a lack of evaluation scores and efficient evaluation indicators for the quality of industrial control code generation of LLMs. Therefore, it is urgent to establish a data set and innovate the scores and indicators of the quality of industrial control code generation of LLMs so as to compare between different LLMs or verify the effectiveness of customized training.

In recent years, the breakthrough progress of large models in the field of code generation has enabled deep learning to show significant advantages in the task of converting natural language to code. Early applications of sequence-to-sequence (Seq2Seq) models[1] showed that deep learning can effectively convert text descriptions into code, significantly reducing the burden on developers. Subsequently, models based on abstract syntax trees (ASTs)[2] further enhanced the grammatical consistency and quality of generated code. The introduction of pre-trained models, such as CuBERT[24] and CodeBERT[25], has achieved good results in multiple programming languages, and GraphCodeBERT[26] has improved code comprehension capabilities by using data flow and structural information. Large-scale pre-trained models such as GPT-3[27] and GPT-4[29] provide strong support for general code generation, but their performance in specific fields such as industrial control still needs to be verified. In addition, researchers explored improvements in prompt engineering[18] and multi-round dialogue generation mode[31], further improving the accuracy of code generation. However, applications in specific fields such as industrial automation still need more exploration, and this article will conduct further research based on existing results.

The contributions of this paper can be summarized as follows:

First, this paper constructs the IC-100 industrial dataset, which addresses the lack of high-quality prompt sets in the field of industrial control code generation. The dataset is built by collecting code requirements from common industrial control scenarios and selecting 100 representative code generation tasks. These prompt tasks cover all core aspects of Python, MATLAB, C++ and IEC 61131-3 Structured Text programming, including I/O handling, PID control, interlocks, diagnostics, communication, and batch control. The IC-100 dataset provides a valuable benchmark for evaluating large language models on practical, industry-specific coding tasks.

Second, based on the IC-100 dataset, the code generation capabilities of large language models such as GPT-4o are evaluated. For each task, the model-generated code is reproduced in actual industrial control development platforms to verify executability. This process ensures that the evaluation is grounded in real-world feasibility, not just theoretical correctness. By reproducing and analyzing the code in software environments, this paper builds a practical code score evaluation framework that reflects the operational and structural quality of the generated code.

Third, this paper proposes a novel evaluation algorithm named ADAP to overcome the limitations of traditional metrics such as BLEU and Pass@k. These conventional metrics rely on fixed parameters (e.g., the fixed n in BLEU or the fixed k in Pass@k), which can be inappropriate for evaluating diverse code generation tasks. ADAP incorporates adaptive algorithms into these metrics, dynamically adjusting evaluation parameters based on task complexity and prompt structure. This significantly enhances the efficiency, sensitivity, and interpretability of the evaluation process, making it more meaningful for industrial applications. The ADAP algorithm not only scores the code output from LLMs but also provides a more nuanced reflection of their practical utility and alignment with industrial requirements.

We input these prompts into mainstream large-scale language models, and analyze the generated code results. We test the generated code through code score evaluation algorithms and evaluation systems such as BLEU, Pass@k, and propose an algorithm called ADAP to improve the existing insufficient evaluation problems. The evaluation results of ADAP algorithm can reflect the quality of code generation. Although existing LLMs still have certain limitations and sometimes generate wrong answers, after the improvement of our ADAP evaluation algorithm, these models can significantly improve development efficiency and improve the running quality of code, at least in the application scenarios of implementing customized code generation.

The rest of this paper is organized as follows: Section 2 outlines the basic logic and preparations. Section 3 presents our experiments, including collection of cue words, scoring of code generation algorithms, and evaluation of large model code metrics. Section 4 presents the results of code generation and code evaluation. Section 5 concludes this paper and proposes deficiencies and future work.

2 Related Work

In recent years, in the field of large-scale model code generation, researchers have extensively explored how to use deep learning models to convert natural language into high-quality code. Early research focused on conversion methods based on sequence-to-sequence models. For example, Ling et al.[1] used recurrent neural networks and attention mechanisms in 2016 to convert natural language descriptions into card codes, thereby reducing the writing burden on developers. Subsequent research further optimized this process. Yin et al.[2] proposed a model based on an abstract syntax tree (AST) in 2017 to better model the grammar of the target language, thereby limiting the generated code format space and ensuring that the generated code complies with the language specification.

In 2019, Sun et al.[16] found that the code structure was significantly different from natural language, and proposed a method based on a tree-based convolutional neural network (Tree-based CNN) to more accurately predict the structure of the abstract syntax tree. Wei et al.[9] designed a multi-task learning framework by taking advantage of the duality of code generation and code summarization tasks, and improved the performance of the model by introducing dual constraints between the two. In 2020, Sun et al.[16] further proposed the TreeGen model based on the Transformer architecture, which enables the model to effectively combine the structural information of the code.

In recent years, the application of pre-trained models in the field of code generation has also made significant progress. For example, CuBERT[24] is the first pre-trained model designed specifically for code generation tasks. It is based on the BERT architecture and uses large-scale Python code for training; the subsequent CodeBERT[25] supports multi-programming language input and is widely used in code generation and understanding tasks. In 2021, Guo et al.[26] proposed GraphCodeBERT, which improves the model's ability to understand the code by introducing the structured information of the code, so that the model can achieve better performance on various downstream tasks.

With the development of generative pre-trained models, the generation quality and usability have been significantly improved. GPT-3[27] released by OpenAI achieves high-quality natural language to code conversion through large-scale parameters, providing a powerful tool for automated code generation. Based on the GPT-3 architecture, tools such as GitHub Copilot[28] have further promoted the application of code generation among developers and significantly improved programming efficiency. With the release of ChatGPT[18] and GPT-4[29], the performance of large models in code generation tasks has been continuously optimized, but its performance in specific fields such as industrial control remains to be further verified.

In order to cope with complex generation tasks, White et al.[18, 19] proposed an improved method of prompt engineering, which enhances the generation effect by introducing specific prompt patterns. This method is also applicable to code generation tasks. The CodeGEN model proposed by Nijkamp et al.[31] in 2023 realizes program generation through multi-round dialogues, allowing users to gradually decompose complex intentions and improve generation quality and usability. Existing research on code generation for large models focuses on generality, while the code generation capabilities in the field of industrial control have not been explored in depth. Our research will further explore the application potential of these models in industrial automation.

In recent years, in the field of programmable logic controllers (PLCs), researchers have devoted themselves to applying large language models (LLMs) to the automatic generation and verification of PLC codes. Fakih et al.[36] proposed the LLM4PLC framework, which uses user feedback and external verification tools to guide LLM to generate PLC code, significantly improving the success rate and quality of code generation. Koziolek et al.[37] explored the application of ChatGPT in PLC and DCS control logic generation and found that LLM can generate grammatically correct IEC 61131-3 structured text code, demonstrating its potential in control engineering. In addition, Cheng et al.[38] developed the G4LTL-ST tool to automatically generate PLC programs from temporal logic specifications, reflecting the application of formal methods in PLC code generation. These studies show that LLM has broad application prospects in the field of PLC code generation, but its performance in specific fields such as industrial control still needs further verification.

3 Methodology

3.1 Experimental Preparation

3.1.1 Implementation Logic

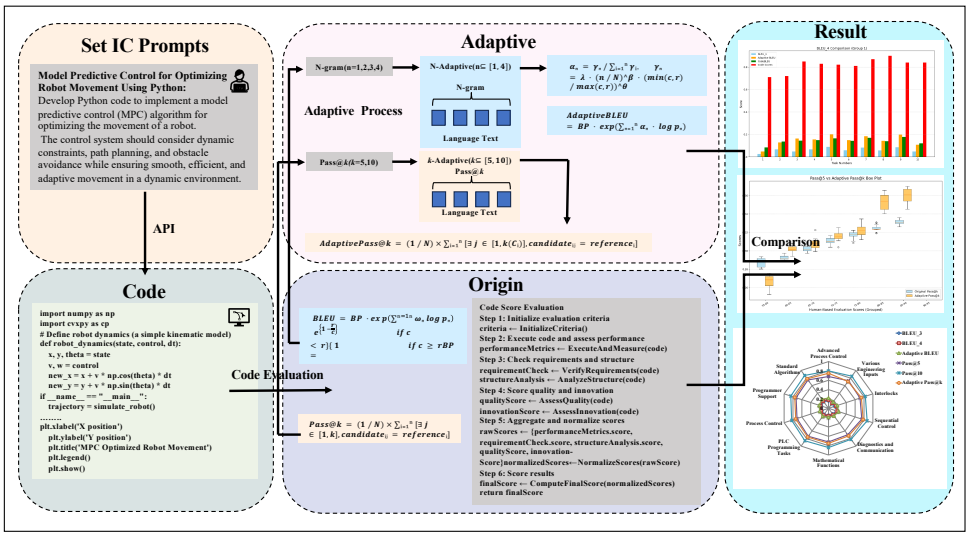

Figure 1: Main Related Work

The figure above shows the overall logical flow of the experiment, which aims to evaluate the effectiveness and accuracy of the PLC code generated based on the large language model (LLM). The main steps of the experiment are as follows:

1. Set IC prompts: According to the experimental environments, prompt information for the industrial control task is used as the input of the model to generate code. The example in the figure is the prompt: Model Predictive Control for Optimizing Robot Movement Using Python

2. Generate code: Generate the corresponding IC code through the large language model (LLM)'API and output the code text. API replys a python code by prompts.

3. Code Score Evaluation: Code score evaluation, BLEU and Pass@k are used to evaluate the generated code.

4. Code comparison: ADAP algorithm are used to compare the code that whether ADAP algorithm can have a better performance than original BLEU and Pass@k.

Finally, by this process , a set of verified prompts and corresponding standard code samples can be generated for further experimental analysis and model optimization.

3.2 Prompt Collection

The prompt words are extracted from real scenarios and common problems in control engineering, including fields such as standard algorithm implementation, process control, sequential control, and advanced process control. The design refers to standard libraries commonly used in industrial control (such as OSCAT Basic library), typical controller models (such as PI and PID control), and actual application cases of batch processing and interlocking logic. To ensure that the prompts can stimulate the potential of large language models, the prompt words describe the tasks through clear domain terms and specific requirements. For example, prompt words involving PLC programming will combine industrial standard language features, while prompts for advanced process control require the model to generate MATLAB, Python, or C++ code. Each prompt is designed with flexibility in mind, and prompts for different scenarios can be generated by adding specific context, modifying the task background, or providing more constraints. In addition, the prompt collection also reserves space for future expansion, allowing researchers to add new task categories or optimize the prompt content.

Many of the prompts are detailed, often including extensive domain-specific instructions. To streamline the presentation, we provide short names in the table, with the full list available online for reference. The prompt collection includes 10 categories, each containing 10 prompts, designed to cover a broad spectrum of control logic engineering tasks.

The prompt collection may be refined and expanded in future work as practitioners and researchers gain a deeper understanding of generative AI. Individual prompts can be enhanced with additional context or replaced entirely to evaluate more specific aspects of an Large Language Model.

We summarize and analyze several of the 100 answers, which are listed in abbreviated form. These 100 data are divided into ten categories, with 10 data in each category.

3.3 Code Score Evaluation

Before using evaluation indicators to judge the quality of code generation, the code generated by the large model is first scored algorithmically. This paper proposes an algorithm to evaluate the quality of code generated by the model based on multi-dimensional criteria and generate a comprehensive score for each piece of code.

3.3.1 Evaluation Steps

Algorithm 1 Evaluation Scoring Process

Step 1: Initialize evaluation criteria
criteria ← InitializeCriteria()
Step 2: Execute code and assess performance
performanceMetrics ← ExecuteAndMeasure(code)
Step 3: Check requirements and structure
requirementCheck ← VerifyRequirements(code)
structureAnalysis ← AnalyzeStructure(code)
Step 4: Score quality and innovation
qualityScore ← AssessQuality(code)
innovationScore ← AssessInnovation(code)
Step 5: Aggregate and normalize scores
rawScores ← {performanceMetrics.score, requirementCheck.score, structureAnalysis.score, qualityScore, innovationScore}
normalizedScores ← NormalizeScores(rawScores)
Step 6: Score results
finalScore ← ComputeFinalScore(normalizedScores)
return finalScore

As shown in the figure above, the scoring process includes the following steps:

First, the system initializes a predefined set of evaluation criteria to ensure consistency across different code samples. This is followed by executing the candidate code and collecting a variety of performance metrics, such as runtime efficiency, memory usage, and correctness under different input scenarios.

Next, the process involves verifying whether the submitted code meets basic requirements and adheres to the expected structure or formatting conventions. This includes checking for the presence of essential functions, proper modularization, and logical flow.

Subsequently, the evaluation focuses on two key qualitative aspects: the overall code quality and the level of innovation. Quality assessment considers factors like readability, maintainability, and adherence to best practices, while innovation scoring examines the novelty of the approach, creative problem-solving, and algorithmic sophistication.

All scores from previous stages are then aggregated into a raw score vector, which is normalized to account for scale differences between various metrics. Finally, a composite final score with sum 1 is computed through a weighted combination or other aggregation method, providing a holistic evaluation of the code's performance, structure, and creativity.

This structured, multi-step evaluation ensures a comprehensive and fair assessment of generated or submitted code across both quantitative and qualitative dimensions.

3.4 Evaluation Metrics

3.4.1 BLEU Series

BLEU (Bilingual Evaluation Understudy) is a commonly used automated evaluation metric for measuring the quality of the output of machine translation or text generation systems. It scores machine-generated translations by comparing their similarity to one or more reference translations, based primarily on the degree of vocabulary overlap and consistency of word order. BLEU scores range from 0 to 1, with higher scores indicating that the generated text is closer to the reference text. This method is applicable to a variety of languages and tasks, and typically uses n-grams to measure translation accuracy, making it an important tool for evaluating the performance of natural language processing models.

$$
\begin{aligned}
BLEU &= BP \cdot \exp\left(\sum_{n=1}^{N} \omega_n \log p_n\right) \\
BP &= \begin{cases} 1, & c \geqslant r \\ e^{\left|1-\frac{1}{c}\right|}, & c < r \end{cases}
\end{aligned}
\tag{1}
$$

In code generation tasks, the code generation process can be regarded as a translation from natural language to code, so BLEU is often used to evaluate the quality of code generation. BLEU is an indicator for measuring the quality of machine translation, which is evaluated by calculating the value of n-gram. n-gram refers to n consecutive tokens in a sentence. The similarity of the generated code is evaluated by calculating the proportion of n-grams in the generated code that appear in the reference code. Usually in code generation tasks, n is 4. In order to ensure the fairness and accuracy of the evaluation, a penalty term needs to be introduced in the calculation of BLEU. The final BLEU value is obtained by multiplying the weighted sum of n-grams by the penalty term. In formula (2), BP represents the penalty term, c represents the length of the model output code, and r represents the length of the reference code.

3.4.2 Pass@k Series

Pass@k is a metric used to evaluate the performance of code generation models, which indicates the probability that at least one of the generated k candidate codes is correct. Specifically, the model generates k different code outputs under given input conditions, and if at least one of them is correct, the model is considered to have "passed" this test. Therefore, Pass@k measures the ability of the model to generate high-quality code. As the value of k increases, the model has a greater chance of generating correct code. This metric is widely used in the evaluation of automatic code generation tasks.

The formula for Pass@k can be defined as, given a set of tasks and k candidate solutions generated for each task, Pass@k represents the probability of successfully solving at least one task:

$$
\text{Pass @k} = \frac{1}{N} \sum_{i=1}^{N} \mathbf{1}(\exists j \in [1, k],\ \text{candidate}\ i, j = \ \text{reference}\ i) \tag{2}
$$

Where:N is the total number of tasks; k is the number of generated candidate solutions; $\mathbf{1}$ is an indicator function whose value is 1 when at least one candidate solution is correct and 0 otherwise; $\text{candidate}_{i,j}$ is the jth candidate solution for the ith

task; reference$_i$ is the reference correct solution for the ith task. The formula shows that Pass@k is the proportion of all tasks for which at least one candidate solution is correct.

3.5 ADAP Algorithm

The BLEU series and pass@k series are commonly used evaluation indicator systems. However, the traditional BLEU series and pass@k series require fixed values for key parameters. For example, in the n-gram of the BLEU series, n is generally fixed (generally from 1 to 4); in the pass@k series, k is generally fixed (generally k=5 or k=10). This will lead to a situation where when n or k takes a certain value, the score may be abnormally low due to the incorrect choice of n or k value. However, such a low score with a fixed n or k value does not necessarily mean that the code generation quality is not good. Based on this situation, we have made adaptive innovations in the algorithms of the traditional BLEU series and pass@k series, and no longer fix n or k values for code evaluation to avoid this situation.

3.5.1 AdaptiveBLEU

The traditional BLEU metric uses fixed n-gram weights, but different tasks pay different attention to n-grams. We can study an adaptive n-gram weight adjustment mechanism to enable the BLEU metric to dynamically adjust weights based on task or sentence characteristics. In this paper, we propose a method to achieve adaptive weight adjustment BLEU.

We dynamically adjust weights for different n-gram levels (such as 1-gram, 2-gram, 3-gram, 4-gram). In general, BLEU uses equal weights to calculate n-gram accuracy by default (1-gram to 4-gram weights are 0.25). To achieve adaptive weight adjustment, these weights can be dynamically adjusted based on sentence length, complexity, or task characteristics. To achieve adaptive n-gram weight adjustment, n-gram weights can be dynamically adjusted based on some specific properties of reference sentences and generated sentences (such as length, complexity, vocabulary richness, etc.). The following is an innovative adaptive weight formula that not only depends on the length of the n-gram, but also combines the sentence length ratio and other complexity indicators. We define a new adaptive n-gram weight formula α_n, whose weight depends on the level n of the n-gram and the sentence length difference.

$$\alpha_n = \frac{\gamma_n}{\sum_{n=1}^{N} \gamma_n}, \quad \gamma_n = \lambda \cdot \left(\frac{n}{N}\right)^{\beta} \cdot \left(\frac{\min(c,r)}{\max(c,r)}\right)^{\theta} \tag{3}$$

Among them, α_n is the adaptive weight of the n-gram. The sum is normalized to 1. γ_n is the basic weight, which controls the contribution of n-gram weights through different parameters. λ is a scaling parameter that adjusts the overall weight and can usually be set to 1. n: The level of the current n-gram, such as 1-gram, 2-gram, etc. N: The level of the maximum n-gram, such as 4 for 1-gram to 4-gram. β: A control parameter for the n-gram level, used to emphasize the influence of high n-grams (larger n) or low n-grams (smaller n). For example, when $\beta > 1$, the weight of high n-grams is increased. $\frac{\min(c,r)}{\max(c,r)}$: The ratio of the candidate sentence length c to the reference sentence length r, used to reflect the difference in length between the two. θ: The weight control parameter of the length ratio. $\theta > 1$ emphasizes the impact of sentence length differences on weight distribution.

When the length difference between the candidate sentence and the reference sentence is large (i.e., $c \neq r$), the long sentence can increase the weight of high n-grams to ensure matching on long phrases. β controls the importance of the n-gram level. For example, when β> 1, it tends to emphasize the matching of high n-grams, which is suitable for code generation tasks and emphasizes the matching of large block structures. θ controls the impact of length differences on weights. If the sentence lengths are very close ($c \approx r$), the weights of each n-gram will be more uniform. When the length difference is large, it can tend to increase the weight of low n-grams and reduce the weight of high n-grams.The BLEU formula with adaptive n-gram weights can be expressed as:

$$BLEU = BP \cdot \exp\left(\sum_{n=1}^{N} \alpha_n \log p_n\right) \tag{4}$$

Where BP is the shorthand length penalty, p_n is the precision of the n-gram, and α_n is the adaptive weight.

3.5.2 AdaptivePass@k

Currently, Pass@k usually uses a fixed number of candidate codes (the value of k is fixed, such as k=5 or k=10), but in actual application scenarios, some tasks may not need to generate so many candidate codes to find the correct solution, while other more complex tasks may require more candidate codes. We propose an adaptive candidate generation strategy that dynamically adjusts the value of k to optimize generation efficiency and resource usage.

We use task complexity and historical data to dynamically adjust the k value (for example, complex tasks automatically increase the number of candidate codes, while simple tasks reduce the k value). Through machine learning or reinforcement learning methods, the model learns when to stop generating more candidate solutions during the generation process. This will reduce the waste of computing resources and improve generation efficiency, while also ensuring the success rate of solving complex tasks.

In terms of formula, we can introduce the task complexity C_i into the formula and let k be a function $k(C_i)$ related to the complexity. In this way, the number of generated candidates can change dynamically according to the complexity of each task. We can define a formula to express the dynamically adjusted Pass@k as follows:

$$\text{AdaptivePass@k} = \frac{1}{N}\sum_{i=1}^{N} \mathbf{1}\left(\exists j \in [1, k\left(C_i\right)], \text{candidate } i, j = \text{ referencei }\right) \tag{5}$$

Where N is the total number of tasks; $k(C_i)$ is the number of candidates dynamically adjusted based on the complexity of the i-th task C_i; **1** is an indicator function that takes the value 1 when at least one candidate solution is correct and 0 otherwise; $\text{candidate}_{i,j}$ is the j-th candidate solution for the i-th task; reference_i is the reference correct solution for the i-th task.

4 Experiments

4.1 Platform and Experimental Environment

This experiment is based on VSCode, MATLAB and CODESYS due to different compile languages in IC-100 dataset. CODESYS (Controller Development System) is an integrated development environment (IDE) widely used for PLC programming based on the IEC 61131-3 standard in industrial automation. It supports multiple programming languages such as structured text, ladder diagrams and function block diagrams, and is able to write, debug and deploy control logic.

4.2 Experimental Analysis

The experimental result table of the code generation part is in Appendix A, which is divided by category, with ten in each category, and can be directly clicked to observe. Since 100 data are too redundant in the display of results, the following operations are performed on the data of the BLEU series and pass@k series. For the convenience of display, 10 types of PLC-ST data are displayed. Each category originally contains 10 data, and these 10 data are weighted averaged. The data obtained after weighted averaging is displayed to ensure the authenticity of the data and the beauty of the chart, which is easy to view. The score table of the code generation evaluation algorithm is in Appendix B, which is also divided by category, with ten in each category, and can be directly observed. Since there are too many pictures generated by the code evaluation indicator experiment, they are placed in Appendix C, which is also divided by category, with ten in each category, and can be directly observed.

First, the code generation evaluation algorithm scores in Appendix B are analyzed. By analyzing the data in the ten categories of tables in Appendix B, except for the two categories of code generation, Process Control and Programmer Support, whose scores are both below 60 points, the scores of the other categories are all above 60 points. This shows that except for the codes generated by Process Control and Programmer Support, which are not trustworthy and still need subsequent manual evaluation, the codes generated by the other categories are trustworthy and can be used as a basis for subsequent development work.

Secondly, the experimental results of the adaptive evaluation algorithm in Appendix C are analyzed. In the experimental results of the adaptive evaluation algorithm, the BLEU series and pass@k series are classic evaluation indicators that existed in the early days. When the code generation evaluation algorithm scores more than 60 points, most of the results of BLEU-n (n=1,2,3,4) and Pass@k (k=5,10) are slightly lower or equivalent to the values of Adaptive BLEU and Adaptive pass@k; when the code generation evaluation algorithm scores less than 60 points, the values of Adaptive BLEU and Adaptive pass@k are much lower. In view of the above two situations, the experimental data, namely the reference code, generated code, and evaluation algorithm, are compared and analyzed, and the following two code generation situations are described in detail.

4.2.1 BLEUseries

When the code generation evaluation algorithm score is greater than 60 points The picture shows the results of the BLEU series of algorithms. The results correspond to the n-gram values of Regular BLEU and AdaptiveBLEU. They correspond to n=1,2,3,4 respectively.

When the value of n in n-gram is 1, the result is not as expected. When the value of n in n-gram is 1, the BLEU-1 score is usually abnormally high because the algorithm only relies on a single word as the matching unit. This is because BLEU-1 only considers the matching rate of unigram frequency and ignores the continuity and contextual relevance of multi-word

sequences, which makes it unable to accurately measure the quality of code generation. Unlike and Adaptive BLEU, these two pay more attention to multi-n-gram matching and semantic consistency, allowing them to provide more representative evaluation results. The high score of BLEU-1 does not match the actual generation quality. It is caused by the limitations of n=1. This situation may mislead the optimization direction of the model in actual evaluation. When the value of n in n-gram is 2, this situation occasionally occurs, but it is significantly less.

Table 1: BLEU_2 Comparison (Group 1) with Change Rate (%)

Task	BLEU_2	Adaptive BLEU	Code Scores	Change Rate (%)
Task 1	0.05	0.06	0.71	16.67%
Task 2	0.13	0.14	0.72	7.69%
Task 3	0.22	0.20	0.86	9.09%
Task 4	0.16	0.18	0.84	12.50%
Task 5	0.20	0.22	0.83	10.00%
Task 6	0.15	0.16	0.81	6.67%
Task 7	0.18	0.19	0.87	5.56%
Task 8	0.14	0.15	0.90	7.14%
Task 9	0.21	0.22	0.84	4.76%
Task 10	0.11	0.12	0.84	9.09%

When the n-gram value of n is 3 or 4, the running results are consistent with the ideal expectations. According to the numerical results in the chart, the BLEU score is slightly lower than AdaptiveBLEU in these cases. This shows that the algorithmic innovation of AdaptiveBLEU not only inherits the advantages of the BLEU series of indicators, but also further improves the ability to evaluate the quality of generated code by dynamically adjusting the weights and matching methods of n-grams. Especially in code generation tasks, AdaptiveBLEU more effectively balances vocabulary matching and semantic consistency, avoiding the limitations of relying solely on fixed n-grams. This also verifies the superiority of AdaptiveBLEU in complex generation tasks, making it a more comprehensive and reliable indicator for measuring code generation quality.

Table 2: BLEU Comparison (Group 3) with Change Rate (%)

Task	BLEU_3	Adaptive BLEU	Code Scores	Change Rate (%)
Task 1	0.22	0.25	0.80	13.64%
Task 2	0.15	0.18	0.89	20.00%
Task 3	0.14	0.17	0.77	21.43%
Task 4	0.24	0.27	0.73	12.50%
Task 5	0.14	0.17	0.87	21.43%
Task 6	0.13	0.16	0.75	23.08%
Task 7	0.13	0.15	0.79	15.38%
Task 8	0.18	0.21	0.86	16.67%
Task 9	0.12	0.15	0.71	25.00%
Task 10	0.11	0.14	0.81	27.27%

Table 3: BLEU_4 Comparison (Group 4) with Change Rate (%)

Task	BLEU_4	Adaptive BLEU	Code Scores	Change Rate (%)
Task 1	0.18	0.22	0.87	22.22%
Task 2	0.14	0.17	0.83	21.43%
Task 3	0.14	0.17	0.83	21.43%
Task 4	0.20	0.23	0.76	15.00%
Task 5	0.22	0.25	0.71	13.64%
Task 6	0.18	0.21	0.70	16.67%
Task 7	0.19	0.22	0.91	15.79%
Task 8	0.16	0.18	0.75	12.50%
Task 9	0.20	0.24	0.88	20.00%
Task 10	0.18	0.21	0.74	16.67%

When the code generation evaluation algorithm score is less than 60 points The picture shows the algorithm results of the pass@k series. Since the data samples of the pass@k series are not as complex as those of the BLEU series, no weighted averaging is performed, and all results are directly displayed. From the chart, it can be seen that no matter when k in the pass@k series is 5 or 10, the value of original_pass@k is slightly lower than that of Adaptive_pass@k. This shows that the Adaptive_pass@k algorithm can also be used for subsequent code evaluation in the industry.

Table 4: BLEU_3 Comparison (Group 6) with Change Rate (%)

Task	BLEU_3	Adaptive BLEU	Code Scores	Change Rate (%)
Task 1	0.19	0.14	0.58	26.32%
Task 2	0.19	0.14	0.58	26.32%
Task 3	0.11	0.06	0.59	45.45%
Task 4	0.17	0.12	0.59	29.41%
Task 5	0.18	0.13	0.57	27.78%
Task 6	0.21	0.16	0.56	23.81%
Task 7	0.21	0.15	0.56	28.57%
Task 8	0.22	0.17	0.56	22.73%
Task 9	0.16	0.11	0.56	31.25%
Task 10	0.12	0.07	0.59	41.67%

4.2.2 Pass@k Series

When the code generation evaluation algorithm score is greater than 60 points The picture shows the algorithm results of the pass@k series. Pass@k directly displays all the results. From the chart, we can see that no matter when k in the pass@k series is 5 or 10, the value of Pass@k (k=5, 10) is slightly lower than that of Adaptive_pass@k. This shows that the algorithm of Adaptive_pass@k can also be used for subsequent code evaluation in the industry.

Table 5: Pass@5 (Group 4) and Code Scores with Change Rate (%)

Task	Original Pass@5	Adaptive Pass@k	Code Score	Change Rate (%)
Task 1	0.66	0.75	0.87	13.64%
Task 2	0.85	0.88	0.83	3.53%
Task 3	0.79	0.82	0.83	3.80%
Task 4	0.67	0.91	0.76	35.82%
Task 5	0.81	0.86	0.71	6.17%
Task 6	0.66	0.97	0.70	46.97%
Task 7	0.63	0.98	0.90	55.56%
Task 8	0.71	0.95	0.75	33.80%
Task 9	0.75	0.78	0.88	4.00%
Task 10	0.85	0.88	0.74	3.53%

Table 6: Pass@10 (Group 8) and Code Scores with Change Rate (%)

Task	Original Pass@10	Adaptive Pass@k	Code Scores	Change Rate (%)
Task 1	0.73	0.76	0.79	4.11%
Task 2	0.88	0.90	0.84	2.27%
Task 3	0.78	0.88	0.88	12.82%
Task 4	0.71	0.75	0.76	5.63%
Task 5	0.94	0.98	0.83	4.26%
Task 6	0.75	0.90	0.80	20.00%
Task 7	0.77	0.91	0.81	18.18%
Task 8	0.94	0.96	0.80	2.13%
Task 9	0.73	0.91	0.75	24.66%
Task 10	0.79	0.81	0.84	2.53%

When the code generation evaluation algorithm score is less than 60 points According to the generated chart, we can also see that for some points of pass@k, the value of Pass@k (k=5, 10) is also quite different from the value of Adaptive_pass@k. For example, in the chart of Pass@k n=5, the value of Pass@k (k=5, 10) in Task 58 is quite different from the value of Adaptive_pass@k. Task 58 corresponds to the PID Temperature Control Gas Turbine in the Process Control category, which also requires writing a section of IEC 61131-3 Structured Text. This also reflects the limitations of large models in the field of industrial control to some extent. After being compensated by the innovative algorithm evaluation, Adaptive_pass@k can also feedback the true quality of the generated code. After being reproduced, the Structured Text is not feasible, and the problem is reflected in time.

Table 7: Pass@5 (Group 6) and Code Scores with Change Rate (%)

Task	Original Pass@5	Adaptive Pass@k	Code Scores	Change Rate (%)
Task 1	0.60	0.58	0.56	3.33%
Task 2	0.79	0.72	0.58	8.86%
Task 3	0.79	0.68	0.59	13.92%
Task 4	0.66	0.63	0.59	4.55%
Task 5	0.86	0.84	0.57	2.33%
Task 6	0.73	0.70	0.56	4.11%
Task 7	0.67	0.64	0.56	4.48%
Task 8	0.74	0.71	0.56	4.05%
Task 9	0.85	0.83	0.56	2.35%
Task 10	0.88	0.87	0.59	1.14%

4.3 Discussion

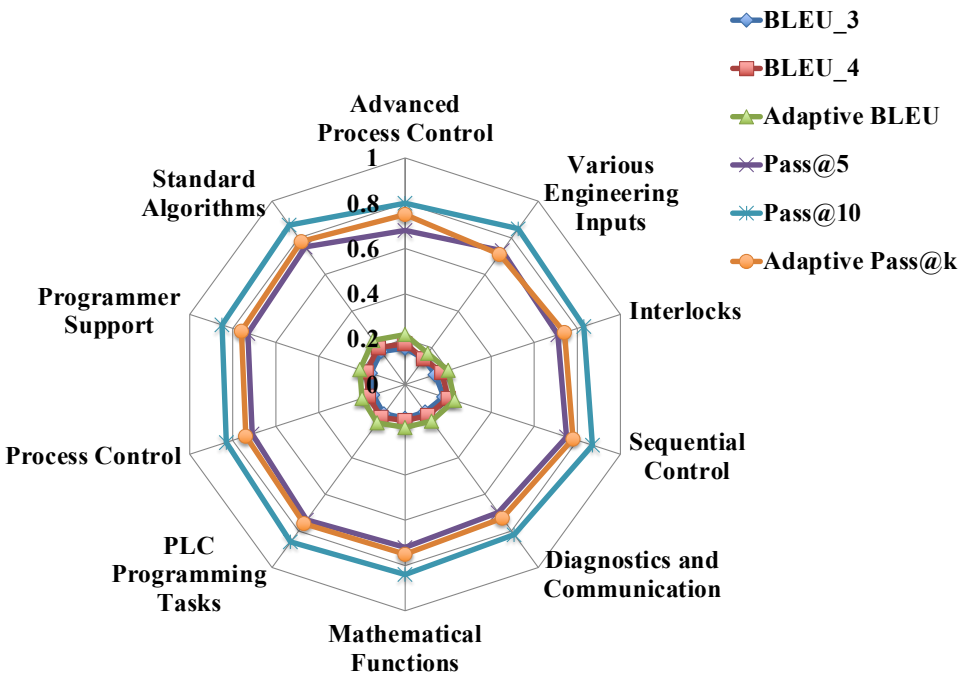

Figure 2: Code Evaluation Radar Chart

As Figure 2 shows, this radar chart presents a comprehensive visualization of code evaluation results across multiple dimensions. The chart includes both technical domains and performance metrics to holistically assess the effectiveness of code generated or evaluated under different conditions.

The inner axes represent task-specific domains, such as PLC Programming, Advanced Process Control, Mathematical Functions, Sequential Control, and Diagnostics and Communication. These axes reflect the functional coverage and capability of the code in handling diverse industrial automation scenarios. The outer axes capture evaluation metrics, including BLEU-3, BLEU-4, Adaptive BLEU, Pass@5, Pass@10, and Adaptive Pass@k. These are commonly used automatic evaluation metrics in code generation research, where BLEU scores reflect linguistic similarity to reference implementations, and Pass@k metrics measure the rate at which correct solutions appear within the top-k generated outputs.

The normalized scale (ranging from 0 to 1) allows for comparative analysis across all metrics and task domains. A more expansive area on the radar indicates stronger performance in both technical breadth and generation quality. This visualization highlights strengths and potential gaps in the evaluated models or algorithms, enabling targeted improvements and balanced benchmarking. Eventually, we caculated that ADAP algorithm has improved 14% averagely compared with original algorithms in the IC-100 dataset.

5 Conclusions and Future Work

The ADAP algorithm can be used as a trusted innovative evaluation algorithm for code evaluation in the industrial control industry. AdaptiveBLEU has a more complete evaluation system than Regular BLEU. From the experimental results, it can be seen that when the code generation evaluation algorithm scores more than 0.6 points, AdaptiveBLEU has a higher evaluation score and is more versatile than BLEU-n (n=1,2,3,4); when the code generation evaluation algorithm scores less than 0.6 points, AdaptiveBLEU will promptly reflect the code problems, and the value of AdaptiveBLEU will be lower than BLEU-n (n=1,2,3,4). AdaptiveBLEU can well make up for the shortcomings of traditional evaluation algorithms. The experimental results show that AdaptiveBLEU can not only cover BLEU-n (n=1,2,3,4)in terms of overall evaluation scores, but also show wide adaptability to different code scenarios, especially when the code is not feasible, timely feedback of problems. This

adaptability manifests itself in a more accurate capture of code complexity, syntactic consistency, and functional accuracy, making it a more effective tool for evaluating code generation tasks.

When the code generation evaluation algorithm score is greater than 0.6, Adaptive_pass@k also has a better evaluation effect than Pass@k (k=5, 10). When the code generation evaluation algorithm score is less than 0.6 points, Adaptive_pass@k will be lower than Pass@k (k=5, 10). The series of algorithms can also still make up for the shortcomings of the traditional evaluation algorithm.

In addition, the versatility shown by AdaptiveBLEU and Adaptive_pass@k shows that these evaluation systems can be applied to a variety of programming languages and complex code generation tasks without major adjustments. This versatility is particularly important in practical applications because it provides a more stable evaluation basis for the expansion of models in different fields. By combining with existing industry evaluation standards, AdaptiveBLEU and Adaptive_pass@k have the potential to become a benchmark for code generation quality assessment. The implementation of these indicators can help developers and researchers more effectively measure the performance of code generation models on a variety of tasks and platforms.

Evaluation metrics such as BLEU and Pass@k have been shown to be very effective in evaluating the quality of generated code, especially in capturing syntactic accuracy and functional correctness. Evaluation systems based on the BLEU series, Pass@k series, etc. can be very good at evaluating the quality of code generation. Therefore, ADAP algorithm can be well applied to code generation in the field of industrial control.

In future work, we will focus on further enhancing LLM's advanced capabilities in handling and generating complex code.Future research can develop along two different tracks. First, enhance LLM's programming capabilities to directly solve a variety of challenging problems, aiming to achieve a performance level of more than 90% of the evaluation platform. Second, explore the use of autonomous agents to achieve more effective collaborative division of labor, which can help solve complex programming challenges more effectively. Through these avenues, we hope to promote the frontier expansion of LLM in the field of code intelligence. In terms of improving LLM's programming capabilities, future research can focus on optimizing fine-tuning techniques to cope with challenging domain-specific problems. This includes introducing a large number of diverse datasets to expand the model's coverage of complex scenarios and minimize reliance on human intervention.

Acknowledgments

This work was supported in part by the National Key Research and Development Program of China (2024YFB4709100), the National Nature Science Foundation of China (61773368), Nature Science Foundation of Liaoning province under(2024-MSBA-82), the Research Program of the Liaoning Liaohe Laboratory(No. LLL23ZZ-04-01), and Liaoning Provincial Science and Technology Plan Project (2023JH26/10100004).

References

[1] W. Ling, P. Blunsom, E. Grefenstette, K. Hermann, T. Kočiský, and F. Wang, "Latent Predictor Networks for Code Generation," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, 2016, pp. 599-609. doi: 10.18653/v1/P16-1057.

[2] P. C. Yin and G. Neubig, "A Syntactic Neural Model for General-Purpose Code Generation," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*, 2017, pp. 440-450. doi: 10.18653/v1/P17-1041.

[3] E. Lai and J. Ma, "Intelligent Uncertainty Handling Using Artificial Neural Networks in a Programmatic Logic Controller-Based Automation System," in *Proceedings of the ASME 2023 International Manufacturing Science and Engineering Conference (MSEC)*, Paper No. MSEC2023-101825, V002T05A003, 8 pages, 2023.

[4] R. W. Brennan and B. McDermott, "Coding Literacy in the Age of Generative AI," in *Service Oriented, Holonic and Multi-Agent Manufacturing Systems for Industry of the Future*, T. Borangiu, D. Trentesaux, P. Leitão, L. Berrah, and J. F. Jimenez, Eds., SOHOMA 2023, Studies in Computational Intelligence, vol. 1136, Springer, Cham, 2024. doi: 10.1007/978-3-031-53445-4_37.

[5] M. Rabinovich, M. Stern, and D. Klein, "Abstract Syntax Networks for Code Generation and Semantic Parsing," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*, 2017, pp. 1139-1149. doi: 10.18653/v1/P17-1105.

[6] O. Ogundare, G. Quiros Araya, and Y. Qamsane, "No Code AI: Automatic generation of Function Block Diagrams from documentation and associated heuristic for context-aware ML algorithm training," *arXiv preprint arXiv:2304.04117*, 2023. Available: https://arxiv.org/abs/2304.04117.

[7] P. C. Yin and G. Neubig, "Reranking for Neural Semantic Parsing," in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, Florence, Italy, 2019, pp. 4553-4559. doi: 10.18653/v1/P19-1447.

[8] X. Jin, J. Larson, W. Yang, and Z. Lin, "Binary Code Summarization: Benchmarking ChatGPT/GPT-4 and Other Large Language Models," *arXiv preprint arXiv:2312.09601*, 2023. Available: https://arxiv.org/abs/2312.09601.

[9] B. L. Wei, G. Li, X. Xia, Z. Y. Fu, and Z. Jin, "Code Generation as a Dual Task of Code Summarization," in *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, 2019, pp. 6563-6573.

[10] I. C. Campbell and I. Bukshteyn, "Putting Knowledge-Based Concepts to Work for Generic Programmable Logic Controller Programming," in *Proceedings of the AAAI Conference on Innovative Applications of Artificial Intelligence (IAAI)*, 1990. Available: https://cdn.aaai.org/IAAI/1990/IAAI90-018.pdf.

[11] G. Koltun, M. Kolter, and B. Vogel-Heuser, "Automated Generation of Modular PLC Control Software from P&ID Diagrams in Process Industry," in *Proceedings of the IEEE*, 2018. doi: 10.1109/SysEng.2018.8544401.

[12] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Mapping Language to Code in Programmatic Context," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, Brussels, Belgium, 2018, pp. 1643-1652. doi: 10.18653/v1/D18-1192.

[13] H. Koziolek and A. Koziolek, "LLM-based Control Code Generation using Image Recognition," *arXiv preprint arXiv:2311.10401*, 2023, 8 pages, 8 figures. doi: 10.48550/arXiv.2311.10401.

[14] Y. Pavlovskyi, M. Kennel, and U. Schmucker, "Template-Based Generation of PLC Software from Plant Models Using Graph Representation," in *2018 25th International Conference on Mechatronics and Machine Vision in Practice (M2VIP)*, Stuttgart, Germany, 2018, pp. 1-8. doi: 10.1109/M2VIP.2018.8600882.

[15] M. Fakih, R. Dharmaji, Y. Moghaddas, G. Quiros Araya, O. Ogundare, and M. A. Al Faruque, "LLM4PLC: Harnessing Large Language Models for Verifiable Programming of PLCs in Industrial Control Systems," *arXiv preprint arXiv:2401.05443v1*, 2024. Available: https://arxiv.org/abs/2401.05443v1.

[16] Z. Y. Sun, Q. H. Zhu, Y. F. Xiong, Y. C. Sun, L. L. Mou, and L. Zhang, "TreeGen: A Tree-Based Transformer Architecture for Code Generation," in *Proceedings of the AAAI Conference on Artificial Intelligence*, 2020, vol. 34, pp. 8984-8991. doi: 10.1609/aaai.v34i05.6430.

[17] N. Liu, Z. Wang, R. G. Baraniuk, and A. Lan, "GPT-based Open-Ended Knowledge Tracing," *arXiv preprint arXiv:2203.03716*, 2022. Available: https://arxiv.org/abs/2203.03716.

[18] J. White, Q. Fu, S. Hays, M. Sandborn, C. Olea, H. Gilbert, A. Elnashar, J. Spencer-Smith, and D. C. Schmidt, "A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT," *arXiv preprint arXiv:2302.11382*, 2023. Available: https://arxiv.org/abs/2302.11382.

[19] J. White, S. Hays, Q. Fu, J. Spencer-Smith, and D. C. Schmidt, "ChatGPT Prompt Patterns for Improving Code Quality, Refactoring, Requirements Elicitation, and Software Design," *arXiv preprint arXiv:2303.07839*, 2023. Available: https://arxiv.org/abs/2303.07839.

[20] R. W. Brennan and J. Lesage, "Exploring the Implications of OpenAI Codex on Education for Industry 4.0," in *Service Oriented, Holonic and Multi-Agent Manufacturing Systems for Industry of the Future*, T. Borangiu, D. Trentesaux, and P. Leitão, Eds., SOHOMA 2022, Studies in Computational Intelligence, vol. 1083, Springer, Cham, 2023. doi: 10.1007/978-3-031-24291-5_20.

[21] A. F. Zambrano, X. Liu, A. Barany, R. S. Baker, J. Kim, and N. Nasiar, "From nCoder to ChatGPT: From Automated Coding to Refining Human Coding," 2023. doi: 10.35542/osf.io/grmzh.

[22] M. Zhang *et al.*, "Towards Automated Safety Vetting of PLC Code in Real-World Plants," in *2019 IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, USA, 2019, pp. 522-538. doi: 10.1109/SP.2019.00034.

[23] H. Koziolek, A. Burger, M. Platenius-Mohr, and R. Jetley, "A classification framework for automated control code generation in industrial automation," *The Journal of Systems and Software*, vol. 166, p. 110575, 2020. Available: www.elsevier.com/locate/jss.

[24] A. Kanade, P. Maniatis, G. Balakrishnan, and K. S. Shi, "Learning and Evaluating Contextual Embedding of Source Code," *arXiv preprint arXiv:2001.00059*, 2020. Available: https://arxiv.org/abs/2001.00059.

[25] Z. Y. Feng, D. Y. Guo, D. Tang, N. Duan, X. C. Feng, M. Gong, L. J. Shou, B. Qin, T. Liu, D. X. Jiang, and M. Zhou, "CodeBERT: A Pre-trained Model for Programming and Natural Languages," in *Proceedings of the 2020 Findings of the Association for Computational Linguistics: EMNLP 2020*, 2020, pp. 1536-1547. doi: 10.18653/v1/2020.findings-emnlp.139.

[26] D. Y. Guo, S. Ren, S. Lu, Z. Y. Feng, D. Tang, S. J. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, and S. Y. Fu, "Graph-CodeBERT: Pre-training Code Representations with Data Flow," *arXiv preprint arXiv:2009.08366*, 2021. Available: https://arxiv.org/abs/2009.08366.

[27] OpenAI, "GPT-3," 2020. Available: https://openai.com/blog/gpt-3.

[28] GitHub, "Copilot," 2023. Available: https://github.com/features/copilot/.

[29] OpenAI, "GPT-4," 2023. Available: https://openai.com/research/gpt-4.

[30] A. Mejia, A. F. Guarnizo, and G. Barbieri, "Assessment of the PLC Code generated with the GEMMA-GRAFCET Methodology," in *Procedia Computer Science*, vol. 200, pp. 2192-2200, 2022. doi: 10.1016/j.procs.2022.01.268.

[31] E. Nijkamp, B. Pang, H. Hayashi, L. F. Tu, H. Wang, Y. Zhou, S. Savarese, and C. M. Xiong, "CodeGEN: An Open Large Language Model for Code with Multi-Turn Program Synthesis," *arXiv preprint arXiv:2203.13474*, 2023. Available: https://arxiv.org/abs/2203.13474.

[32] Z. Yang, S. S. Raman, A. Shah, and S. Tellex, "Plug in the Safety Chip: Enforcing Constraints for LLM-driven Robot Agents," *arXiv preprint arXiv:2309.09919*, 2023. Available: https://arxiv.org/abs/2309.09919.

[33] D. Kanade, P. Maniatis, and G. Balakrishnan, "CuBERT: Contextual Embedding of Source Code," in *Proceedings of the 2023 IEEE Conference on Neural Information Processing Systems*, 2023.

[34] W. Yan, H. Liu, Y. Wang, Y. Li, Q. Chen, W. Wang, T. Lin, W. Zhao, L. Zhu, H. Sundaram, and S. Deng, "CodeScope: An Execution-based Multilingual Multitask Multidimensional Benchmark for Evaluating LLMs on Code Understanding and Generation," *arXiv preprint arXiv:2311.08588*, 2023. Available: https://arxiv.org/abs/2311.08588.

[35] T. Zhang, V. Kishore, F. Wu, K. Q. Weinberger, and Y. Artzi, "BERTScore: Evaluating Text Generation with BERT," *arXiv preprint arXiv:1904.09675*, 2019. Available: https://arxiv.org/abs/1904.09675.

[36] M. Fakih, M. A. Hameed, J. F. Oteafy, and K. Barker, "LLM4PLC: A Framework for Generating Verifiable PLC Code Using Large Language Models," *arXiv preprint arXiv:2401.05443*, 2024. Available: https://arxiv.org/abs/2401.05443.

[37] H. Koziolek, A. Obermaier, M. Ritz, and L. Schwabe, "Using ChatGPT for Industrial Automation Engineering: A Preliminary Study on PLC and DCS Programming," *arXiv preprint arXiv:2305.15809*, 2023. Available: https://arxiv.org/abs/2305.15809.

[38] C. Cheng, C. Buckl, A. Knoll, and S. Kugele, "G4LTL-ST: Automatic PLC Code Generation from Timed Temporal Logic Specifications," *arXiv preprint arXiv:1405.2409*, 2014. Available: https://arxiv.org/abs/1405.2409.