Generating PLC Code with Universal Large Language Models

Kilian Tran, Jingxi Zhang,
Jérôme Pfeiffer, Andreas Wortmann
University of Stuttgart
Stuttgart, Germany
{firstname.lastname}@isw.uni-stuttgart.de

Bianca Wiesmayr
LIT CPS Lab, Johannes Kepler University Linz
Linz, Austria
{firstname.lastname}@jku.at

2024 IEEE 29th International Conference on Emerging Technologies and Factory Automation (ETFA) | 979-8-3503-6123-0/24/$31.00 ©2024 IEEE | DOI: 10.1109/ETFA61755.2024.10711113

***Abstract*—Control software for production systems is typically developed by domain experts, despite its high complexity. The increasingly available Large Language Models (LLMs) can assist developers with code generation and debugging. However, their suitability for generating control software for production systems is still unexplored. Therefore, this study explores the generation of Structured Text (ST) according to IEC-61131-3 by different LLMs. We selected 21 coding examples that are representative of PLC programming and developed an approach for comparing the outputs of different LLMs using metrics for testing generated code (CodeBERTScore, pass@k, generation time). The strategies for prompt optimization that were developed as part of this work can be directly used for improved ST generation. Our results show that, at the time of the study, ChatGPT-4 had the highest reliability in generating syntactically correct ST code that expresses the desired functionality.**

***Index Terms*—Code Generation, Structured Text, IEC 61131, Large Language Model**

I. Introduction

Automation and digitization are vital for maintaining competitiveness in today's production industry. Typically, this happens through Programmable Logic Controllers (PLCs), which act as an interface between the digital and the physical world [1]. PLC programming differs significantly from programming with modern programming languages as it tightly interacts with the hardware components, leading to strict timing requirements. Typical challenges are the design of reliable software that can manage changes in the requirements and safety regulations that demand high-quality standards [2]. As automating a cyber-physical system requires knowledge of the hardware and processes that are involved, control software is typically developed by domain experts [3], i.e., stakeholders with little formal software engineering training. As production systems are tailored to customer's needs and are typically one-of-a-kind, the software engineering effort is relatively high. Having to manually create PLC code, consequently, is a subpar use of the production experts' time and might lead to subpar code quality [4] as well. Automating the creation of PLC code can help to save time and, thus, shift the focus to reusable, maintainable, and correct code. Generative AI, in the form of large language models (LLMs), promises to automate the creation of source code [5]. LLMs may also enable generating PLC code from natural language specifications. Successfully generating useful PLC code will require an understanding of which LLM to employ, and how to optimize the input specification. We focus on Structured Text (ST), which is a textual language defined in the IEC 61131-3 standard. This standard defines five programming languages for PLCs. ST is one of the most popular PLC programming languages and most closely resembles a traditional programming language [6]. Therefore, we investigate the use of LLMs that have been trained on general corpora and have been shown to successfully generate general-purpose programming language (GPL) code. Our study yields a better understanding of the potential benefits and problems of using general LLM-based code generators. In the following, Sec. II discusses related research and Sec. III describes LLMs and their comparison. Sec. IV outlines the reusable evaluation method. Afterward, Sec. V presents our results and Sec. VI discusses these and the evaluation method. Sec. VII concludes.

II. Related Work

Generally, using AI-based code generators has shown to be beneficial in professional environments by enhancing productivity and efficiency [5]. This success has led to specifically tailored LLMs for generating GPL code, such as Github Copilot [7] or Code Llama [8].

A. Model-based PLC Code Generation

Automated generation of PLC code has been subject of research for decades, an overview and classification are provided in [9]. All approaches require a high-level specification as an input for a transformation. For example, Julius et. al. [10] generated ST code from Grafcet models, which are used in automation engineering as a behavior description for systems. The approach retains hierarchical structures present in the Grafcet model to ensure the maintainability of the ST code. Another work introduced a method for automatically generating ladder diagrams (LD) from a finite state machine (FSM) [11]. In the FSM, states represent activated signals and transitions represent corresponding signal changes. Each state can be associated with an output and from this, LD can be generated with each state representing a circuit [11]. Similarly, Sequential Function Charts and ST code have been generated from UML models (e.g., [12]). A limitation of these methods

Authorized licensed use limited to: Universite de Rennes 1. Downloaded on March 08,2025 at 09:32:34 UTC from IEEE Xplore. Restrictions apply.

is that an additional methodology for specifying the expected behavior of PLC code needs to be available or learned. With LLM-based PLC code generation, the desired functionality can be expressed using natural language.

B. LLM-based Code Generation

LLMs were originally developed to mimic and generate natural language [13]. However, the question arises whether this technology is also adequate for generating code. A comprehensive study investigates using ChatGPT 3.5 to generate source code in ten different target GPLs, but not PLC programming languages. [14]. The study suggests that the suitability depends on the target GPL, with high-level dynamically typed languages performing better than lower-level statically typed languages. Out of the 4000 runs, 45.8% of the generated code is successfully executable, with Julia (81.5% executable) performing the best and C++ the worst (7.3% executable) [14].

C. LLM-based PLC Code Generation

At present, only few studies have investigated the suitability of ChatGPT for PLC code generation. Koziolek et al. [15] assessed the capability of ChatGPT-4 to generate ST code. They evaluated their study with 100 different prompts that examined the output of various prompts for syntactic and functional correctness using a subjective scoring. The majority of the output exhibited correct syntax and the code samples appeared robust and efficient. The quality of generated control code can be improved through Retrieval-Augmented Generation, which provides the LLM with sufficient information to include domain libraries in generated code [16]. Boudribila et al. [17] evaluated the use of ChatGPT (exact version not specified) for PLC programming based on a single example. The study evaluated the LLM's capabilities for identifying relevant components, generating code, and checking code against its requirements. ChatGPT was able to identify entities and actions from natural language descriptions. However, both studies only covered ChatGPT, rather than comprehensively comparing different LLMs. Our study extends this body of knowledge by comparing different LLMs to generate ST for production systems.

III. Background

We present available versions of LLMs, as well as metrics for comparing LLMs. The choice of LLMs has the requirement of being able to run locally without expensive hardware, such as a Nvidia A100.

A. LLMs under Investigation

We have included OpenAI's ChatGPT[1] (utilizing GPT-3.5 and GPT-4), Google's Bard[2] (2023.12.18), Meta's Code Llama[3] (7B + 4 Bit Quantization), Platypus2[4] (13B + 4 Bit Quantization), and StabilityAI's StableCode[5] (3B + 5 Bit Quantization) in our study. It is unclear whether the presented language models have been trained on PLC code.

1) ChatGPT: Both versions are based on the transformer architecture, designed for natural language understanding and generation[1]. Trained on extensive text corpora and fine-tuned with human feedback, GPT-4 has excellent coding abilities for GPLs and the capacity to comprehend visual inputs[1].

2) Code Llama: Code Llama is a specialized LLM that was finely tuned for code generation and is based on the Llama2 model[3]. It was extensively trained on code-specific datasets. It demonstrates impressive coding capabilities, capable of generating and debugging GPL program code[3].

3) Bard: Bard is based on Google's LLM PaLM2, designed for understanding, summarizing, and generating texts[2]. Trained on diverse corpora, Bard is still in the experimental phase, subject to updates and changes.

4) Platypus2: Platypus2, developed by Boston University, is an LLM focused on acquiring logical knowledge, relying on the Open-Platypus dataset.[4] While demonstrating significant performance in benchmarks, its ability for code generation remains unexplored.[4]

5) StableCode: StableCode, a code-specific LLM trained on the Stack Dataset of BigCode, distinguishes itself with its relatively small size (three billion parameters)[5]. Tailored for programming assistance, it holds potential advantages for compatibility with consumer PCs.

B. Quantization

Executing LLMs requires powerful and modern hardware [18]. The required memory, in bytes, is approximately four times the number of parameters. One way to address this is quantization. The parameters of an LLM, representing the weights and biases, are typically represented as 32-bit floating-point numbers. The idea of quantization is to convert these numbers from higher precision to lower precision, such as 16-bit float, 8-bit integer, or 4-bit integer. This reduces the memory requirements and optimizes computational performance. However, a drawback is the lower precision [19]. This study uses quantized models of Code Llama, Platypus2, and StableCode due to the following hardware constraints: AMD Ryzen 7 4800h eight-core processor á 4,2 GHz with 16 threads; 16 GB RAM; Windows 11; Python 3.10.6.

C. Metrics

The metrics CodeBERTScore and pass@k are used because they are specifically developed to test machine-generated code. In addition, the time taken by the LLM to generate the ST code is also measured. Evaluation metrics and benchmarks now exist to test the capability of code synthesis using LLMs. However, most testing tools are tailored to higher-level programming languages like Python, therefore, they cannot be used for this study [20]. This study focuses on evaluating whether the code is executable. Quality metrics, such as the cognitive complexity of generated ST code [21], are not considered.

[1] ChatGPT: chat.openai.com
[2] Google Bard: bard.google.com
[3] Llama: huggingface.co/TheBloke/CodeLlama-7B-Instruct-GGUF
[4] Platypus2: huggingface.co/TheBloke/Platypus2-13B-GGUF
[5] StableCode: huggingface.co/TheBloke/stablecode-instruct-alpha-3b-GGML

Authorized licensed use limited to: Universite de Rennes 1. Downloaded on March 08,2025 at 09:32:34 UTC from IEEE Xplore. Restrictions apply.

1) CodeBERTScore: The CodeBERTScore [22] introduces an evaluation metric for machine-generated code that measures the similarity between the generated code and a reference code. The advantage of this metric lies in its simple calculation and its good assessment of the syntactic structure of programs. CodeBERTScore also considers semantic similarity to the reference code [22] using the pre-trained CodeBERT model [23]. The similarity is calculated by representing tokenized code as vectors and computing pairwise cosine similarity This calculation allows for the creation of a similarity matrix. From this matrix, $Precision, Recall, F_1$, and F_3 can be computed [22]. The $Precision$ indicates how many tokens are translated/generated correctly, while $Recall$ indicates how many of the truly correct tokens are also generated [24]. In other words, $Precision$ represents accuracy, while $Recall$ represents the hit rate. The F_1-Score is the harmonic mean between $Precision$ and $Recall$. The F_3-Score also considers $Precision$ and $Recall$, but with a higher weight on $Recall$. For this study, the F_3 is more important, as $Recall$ provides more insight into how closely the generated code matches the reference code [22], [24]. While these comparison metrics are unable to evaluate the functionality of the code, a correlation between a high rating and correct functionality has been observed [22].

2) pass@k: The pass@k indicates the probability that at least one of the top-k generated code samples exhibits correct functionality [25]. Traditional token-matching metrics like BLEU have limited applicability, as they are not well-suited to assess the functionality or executability of the code. The pass@k score addresses this issue and is a common metric for evaluating the performance of LLMs in coding, allowing for comparisons with other models.

IV. Methodology and Implementation

We followed the evaluation method as depicted in Fig. 1. Firstly, the LLMs are evaluated solely based on their ability to generate IEC-61131-3 ST. This decision was made because most LLMs generate textual outputs, simplifying the evaluation and assessment of the code. Each LLM receives the same prompt to allow comparison. We used the metrics CodeBERTScore and pass@k to compare the output by the evaluated LLMs (*cf.* Sec. III-C). Furthermore, Codesys V3.5 SP19 Patch19 [26] is used to test the functionality of the generated programs. Codesys is a widespread tool that is free-to-download, supports the IEC 61131-3 standard, and hardware-independent [6]. Hence, it is used in this study.

A. Prompts

Using prompts, instructions are provided to the LLM to generate ST. For evaluation, a collection of 21 prompts has been compiled to test ST generation. These prompts cover various aspects of ST programming, such as the use of IF-ELSE branches, FOR loops, SWITCH-CASE statements, as well as the utilization of timers and the ability to implement functions. Furthermore, a more extensive program is generated to test the AI's capabilities in handling such tasks.

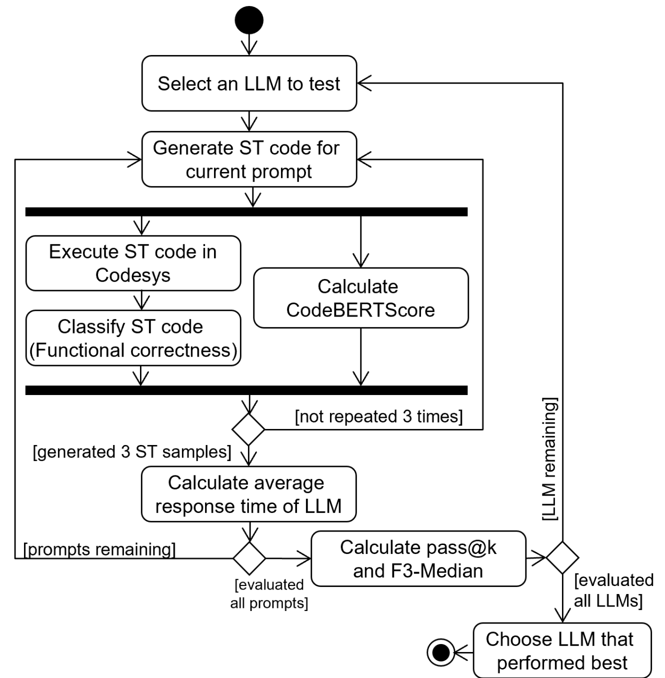

Fig. 1: Activity diagram illustrating the evaluation process. It was performed before and after prompt optimization.

Each prompt is accompanied by a reference code used for calculating the CodeBERTScore. The collection consists of six mathematical functions, five process control functions, seven processes, one interlock, and two calculations. The reference code was sourced from publicly available examples, that we include in our replication package.[6] The prompts are formulated in detail to best test the LLMs and contain explicit instructions for generating IEC-61131-3 ST. The presence of the keywords `BEGIN`, `START`, `PROGRAM <Name>`, `END_PROGRAM`, `METHOD <Name>`, and `END_METHOD` is irrelevant since they are not essential for program execution.

B. Evaluation of Generated Code

Each LLM gets the 21 prompts as the input to generate 21 ST codes, with the time required for generation being measured. Every prompt is composed with the generic instruction `Give me an IEC 61131-3 structured text program`, followed by the functionality of the program. All PLC programs are checked for syntactic and semantic correctness. For syntactic validation, similarity to the reference code is computed using the CodeBERTScore. Semantic correctness is tested by compiling and executing the logic in Codesys, categorizing the code into one of four categories: *FAIL*, *NONEX*, *INCOR* and *PERF*. Four categories are defined for classifying ST code:

- *FAIL*: No code generated, i.e., the model generates text that does not represent ST.

[6]Replication package: https://github.com/iswunistuttgart/generating_plc_code_with_llms/tree/main

Authorized licensed use limited to: Universite de Rennes 1. Downloaded on March 08,2025 at 09:32:34 UTC from IEEE Xplore. Restrictions apply.

- *NONEX*: Non-executable code that has ST syntax, but is not syntactically correct according to Codesys.
- *INCOR*: ST code with incorrect functionality that compiles, but fails the test cases.
- *PERF*: The ST code compiles and satisfies the test cases.

After generating the 21 ST codes, the process is repeated three times for the same LLM. On the one hand for the calculation of the pass@3 score, and on the other hand for consistency checks. After generation, the median of the F_3 score and the average time from the three runs are calculated. Then, the average of these values from the 21 sub-results is determined. The pass@k is the primary metric followed by the F_3 score for selecting the best LLM.

Furthermore, the entire evaluation is conducted twice. In the first run, ST is generated with unchanged prompts. Based on the insights from this run, the prompts are augmented and optimized, a process known as *Prompt Engineering*. It involves designing and formulating prompts to achieve improved results [13]. The original collection, *i.e.,* without optimization, already has a detailed description of functionality. In the second run, explicit points that the LLM should avoid are highlighted, depending on the resulting errors the LLMs produce. An example could be `Use IF-ELSE branches instead of Switch-Cases`. Additionally, the prompt specifies which variables to use, as in practice, the programmer is likely to want to use specific input and output devices. An example could be: `Use following variables: I1: BOOL; I2: BOOL; I3: REAL; O1: BOOL;` The new prompts include the prompt optimization (the rules to avoid errors), the prompt that describes the specification of the program, and the desired variables to use.

C. Implementation

The implementation of our process utilizes the LLM API to generate three sets of 21 PLC logics and calculates their similarity to reference code using the CodeBERTScore library. The developer executes the generated programs with the Codesys component for categorization. Some aspects of the analysis are automated. Manual code generation is performed for ChatGPT and Bard due to API costs or unavailability, while local LLMs automate code generation. Verifying the executability and correctness of generated PLC logic poses a challenge. Unfortunately, no functional automated testing solution was identified, necessitating manual examination of programs using Codesys. The implementation utilizes the libraries `ctransformers`, `code_bert_score`, `pandas`, and `numpy`. For generating STs our implementation iterates over each prompt and passes it to the respective LLM.

During the generation, the time is measured, and the generated ST is then saved in a file. This process is repeated three times. For calculating the CodeBERTScore our implementation iterates over each generated ST and reference ST including the corresponding prompt. Furthermore, we calculated the pass@k-Score according to [25].

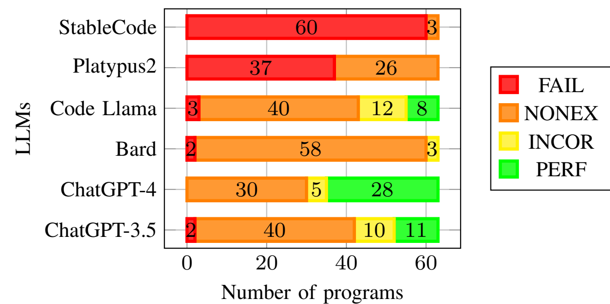

Fig. 2: Correctness of generated programs (without prompt optimization)

V. Evaluation Results

In this chapter, we first present the common syntax errors in the two runs for each LLM. Then, we compare the output based on the metrics.

A. Performance without Prompt Optimization

Generated code was categorized into four groups according to Sec. IV-B in Fig. 2. StableCode produces only three executable ST programs, while Platypus2 exhibits ST-like syntax in 26 programs, with few exceptions. For the remaining LLMs, most generated programs are ST. Among them, ChatGPT-4 generates the most functional PLC programs (28), followed by ChatGPT-3.5 (11) and Code Llama (8). Only 5 out of 63 programs generated by Bard are executable. In total, only 30 of the generated programs contain incorrect functionality. Interestingly, the code generated by locally running LLMs is similar, sometimes identical, in all three attempts.

1) Common Mistakes: Common errors as depicted in Table I. The frequency of errors is classified into three categories: frequently, occasionally, and rarely. Frequent errors occur more than 50% of the time, occasional errors occur in about 25% of cases, and rare errors occur only a handful of times. Whereas the errors for incorrect syntax, usage of `printf`-functions and incorrect usage of `RETURN`-Statements are not severe, fixing them requires high manual effort. On the other hand, errors for non-ST or timer tend to become more time-consuming to fix. The errors encountered are mainly syntactic, and StableCode shows the most errors. Most programs generated by this model either show JavaScript-like syntax or are plain text. In contrast, only half of Platypus2's generated programs have ST-like syntax. The other models mostly produce correct ST structures. Frequent errors, particularly in locally executed LLMs, are related to the syntax of control structures. For example, `ELSIF` is often written as `ELSE IF`, or the control structures are not properly terminated. In some cases, the `END_<control structure>` is missing. Additionally, LLMs encounter issues with switch cases. ChatGPT treats the cases as conditions in the `IF-ELSE` branches, which does not conform to the syntax of switch cases. In some models, such as StableCode and Code Llama, the variable declaration is faulty due to missing the correct variable

Authorized licensed use limited to: Universite de Rennes 1. Downloaded on March 08,2025 at 09:32:34 UTC from IEEE Xplore. Restrictions apply.

TABLE I: Common mistakes without prompt optimization

Errors	Stable-Code	Platy-pus2	Code Llama	Bard	GPT 3.5	GPT 4
No ST	●	◑	○	○	○	○
`IF-ELSE` Syntax Error	●	●	◑	○	○	○
Switch-Case Syntax Error	○	○	○	●	●	●
Incorrect termination of control structures	●	◑	◑	○	○	○
Incorrect Variable declaration	●	●	◑	●	◑	○
Usage of Print-Funktion	○	◑	○	●	○	○
Incorrect Usage of `RETURN`-Statements	●	●	●	●	◑	◑
Incorrect Usage of Timer	●	●	●	●	◑	○
● frequently	◑ occasionally		○ rarely/not the Case			

declaration. Furthermore, in Bard, `END_VAR` is frequently omitted. Another common error involves the use of `printf`-functions, which do not exist in PLC programming. All models struggle with the `RETURN` statement in functions. In ST, the `RETURN` statement is not used to return parameters, but to return to the main program. This mistake is made by all LLMs. Most models also face difficulties in correctly applying timers during initialization or retrieval. Bard frequently misplaces code comments in various programs, causing entire code blocks to be commented out. On the basis of the insights gained, prompts can be optimized to effectively minimize these errors. Addressing the most common errors, the following additional instructions for prompts have emerged:

Following rules should be followed:

1) Variables should be declared between `VAR` and `END_VAR`;
2) When using control structures do not forget; `END_<control structure>`;
3) Avoid switch cases unless instructed in the task;
4) Use the right quotation marks for the string type;
5) Implementing a function should be avoided;
6) Only when implementing a function: the variables should be declared inside the function; the return value should be stored in a variable named after the function; `RETURN` should be used correctly, to return to the main method and not to return a value;

In addition, relevant variables will be added, considering that in realistic scenarios, the developer typically intends to use specific input and output devices.

B. Performance with Prompt Optimization

We observe in Fig. 3 that functionally correct ST is generated more frequently. Variable declarations were not considered anymore, as the optimization requires that the user has provided them to the LLM. Approximately half of the

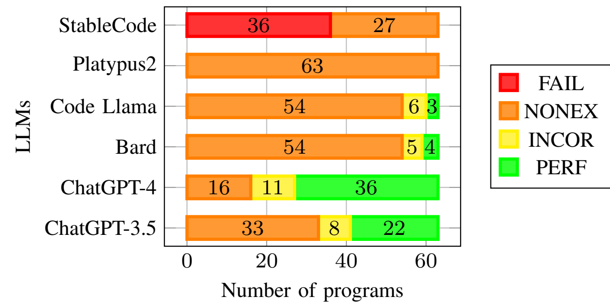

Fig. 3: Correctness of generated programs (with prompt optimization)

TABLE II: Common mistakes with prompt optimization

Errors	Stable-Code	Platy-pus2	Code Llama	Bard	GPT 3.5	GPT 4
No ST	◑	○	○	○	○	○
`IF-ELSE` Syntax Error	●	◑	○	◑	○	○
Switch-Case Syntax Error	●	◑	●	●	◑	◑
Incorrect termination of control structures	●	◑	○	○	○	○
Usage of Print-function	◑	●	○	●	○	○
Incorrect Usage of `RETURN`-Statements	●	●	●	●	◑	○
Incorrect Usage of Timer	●	●	●	●	◑	○
● frequently	◑ occasionally		○ rarely/not the Case			

generated programs have C-like syntax. Platypus2 was capable of producing programs in ST syntax, but they remained non-executable and typically showed syntactic errors. Interestingly, Code Llama produced fewer correct programs after prompt optimization. However, the majority of the generated programs are non-executable. Bard was better able to generate correctly executable code, but only for 4 programs of the first run, and 9 programs over all three runs.

ChatGPT-3.5 is capable of delivering performance similar to that of ChatGPT-4 without prompt optimization. Again, ChatGPT-4 achieved the best performance of all LLMs, with over half of the generated PLC logics being correctly executable. Common errors for prompt optimization are summarized in Table II. In general, we observed fewer errors. StableCode, Platypus2, and Bard failed to terminate the syntax of control structures correctly despite explicit instructions. Furthermore, switch-case-statements were used incorrectly in most cases for all LLMs. In particular, in three ChatGPT-4 samples, the syntax for case conditions is correct, but the instructions are placed between `BEGIN` and `END`, leading to a compiler error. The problem of misusing the `RETURN` statement persists. All language models, except for ChatGPT-4, use `RETURN` to return parameters. ChatGPT-4 is also the only model that correctly implements the application of a timer. An issue arises with the generation of long programs, such as

Authorized licensed use limited to: Universite de Rennes 1. Downloaded on March 08,2025 at 09:32:34 UTC from IEEE Xplore. Restrictions apply.

TABLE III: The average of the medians of $Precision$, $Recall$, $F1$, and $F3$ from the three runs without prompt optimization

Model	$Precision$	$Recall$	$F1$	$F3$
Bard	0.74984	0.80241	0.77391	0.79632
ChatGPT-3.5	0.74306	0.79420	0.76608	0.78814
ChatGPT-4	0.74148	0.80821	0.77216	0.80051
Code Llama	0.77725	0.79300	0.78374	0.79093
Platypus2	0.72155	0.69657	0.70749	0.69852
StableCode	0.68841	0.67396	0.68017	0.67504

TABLE IV: The average of the medians of $Precision$, $Recall$, $F1$, and $F3$ from the three runs with prompt optimization

Model	$Precision$	$Recall$	$F1$	$F3$
Bard	0.86187	0.86826	0.86392	0.86721
ChatGPT-3.5	0.84051	0.87374	0.85479	0.86954
ChatGPT-4	0.80420	0.86744	0.83317	0.86010
Code Llama	0.85150	0.86328	0.85631	0.86171
Platypus2	0.81878	0.83539	0.82628	0.83343
StableCode	0.75569	0.79781	0.77507	0.79297

one prompt, which contains 200 lines of code. Specifically, local LLMs fail to generate code after 70 lines of code, repeating certain keywords. A possible cause identified in previous research [27] is the limited attention span of LLMs, which causes the language model to lose context after a certain number of tokens. In Gemini, further generation needs to be explicitly requested by entering another prompt.

C. CodeBERTScore

For this analysis, the median of the $F3$ score from the three runs is selected. Subsequently, the average of the 21 medians is calculated. In Table III, these values are presented for the run without prompt optimization, and in Table IV, with prompt optimization. For this work, the $F3$, as explained in Sec. III-C1, provides insight into how closely the generated code resembles the reference code. As evident in Table III, ChatGPT-4 performs the best in the first run. Bard, ChatGPT-3.5, and Code Llama also achieve around 80% similarity. Significantly lower are Platypus2 and StableCode with 69.85% and 67.5%, respectively. Most samples from StableCode and Platypus2 exhibit C-like syntax. Concerning the other parameters, it is evident that StableCode performs the worst. Interestingly, Code Llama has the highest $Precision$ and $F1$ score. Code Llama is also capable of generating ST-like code. However, ChatGPT-4 has the highest $Recall$.

In the second run, ChatGPT-3.5 performs best with a similarity of 86.95%, as shown in Table IV. Surprisingly, ChatGPT-4 ranks fourth, despite generating the most correctly executable logics. Once again, StableCode ranks at the bottom in all four metrics with a similarity of just under 80%. Bard achieves the highest $Precision$, ChatGPT-3.5 has the highest $Recall$, while Bard has the highest $F1$ score. Moreover, we observe that the average $F3$ score has increased from 75.82% to 84.75% through prompt optimization. This indicates that precise prompts and specific instructions potentially lead to higher code quality. The similarity to each reference code is shown in Fig. 4. It becomes evident that during the first run, all language models achieve a consistent similarity, but in the second run, especially for six of nine prompts the similarity is lower than the rest. The reason could be that these prompts are input prompts for functions.

TABLE V: pass@k-Score for k=1,2,3 without prompt optimization

Model	pass@1	pass@2	pass@3
Bard	0,0%	0,0%	0,0%
ChatGPT-3.5	17,46%	32,1%	44,35%
ChatGPT-4	44,44%	69,53%	83,52%
CodeLlama	12,7%	23,96%	33,94%
Platypus2	0,0%	0,0%	0,0%
StableCode	0,0%	0,0%	0,0%

TABLE VI: pass@k-Score for k=1,2,3 with prompt optimization

Model	pass@1	pass@2	pass@3
Bard	6,35%	12,39%	18,14%
ChatGPT-3.5	34,92%	58,01%	73,16%
ChatGPT-4	57,14%	82,03%	92,63%
Code Llama	4,76%	9,37%	13,83%
Platypus2	0,0%	0,0%	0,0%
StableCode	0,0%	0,0%	0,0%

D. pass@k

The pass@k-score (Section III-C2) indicates the probability that at least one of the top k generated programs is functional. Table V summarizes the probabilities without prompt optimization. We observe that ChatGPT-4 shows the best performance, where almost every second generated PLC program is perfect in the first attempt. With three attempts, the probability rises to 83.52%. With a probability of 17,46% ChatGPT-3.5 comes in second place. Code Llama has only a 12.7% chance of generating correct PLC programs. Bard, Platypus2, and StableCode were unable to generate correct ST (0,0%). As mentioned in Sec. V-A1, Platypus2 and StableCode mostly generated C code or non-ST. In cases where ST was generated, the programs could not be compiled. Bard, on the other hand, is capable of generating non-executable ST code. Prompt optimization improves performance, as seen in Table VI. The probability increases by 28.58% for ChatGPT-4 and 100% for ChatGPT-3.5. ChatGPT-4 exhibits the best performance here, reaching a probability of 92.63%. Surprisingly, the probability for Code Llama has decreased to 4.76%. On the other hand, with prompt optimization, Bard is capable of generating functional ST and achieves 6.35% in pass@1. Platypus2 and StableCode remain unable to generate the correct executable code.

E. Time

A detailed overview of execution time for each LLM with and without prompt optimization is included in Fig. 5. On average, Google's Bard required the least time, while Platypus2 took the longest. For Bard, ChatGPT-3.5, and ChatGPT-4, the respective services were used, causing a reliance on computation load and connection quality. Surprisingly, prompt

Authorized licensed use limited to: Universite de Rennes 1. Downloaded on March 08,2025 at 09:32:34 UTC from IEEE Xplore. Restrictions apply.

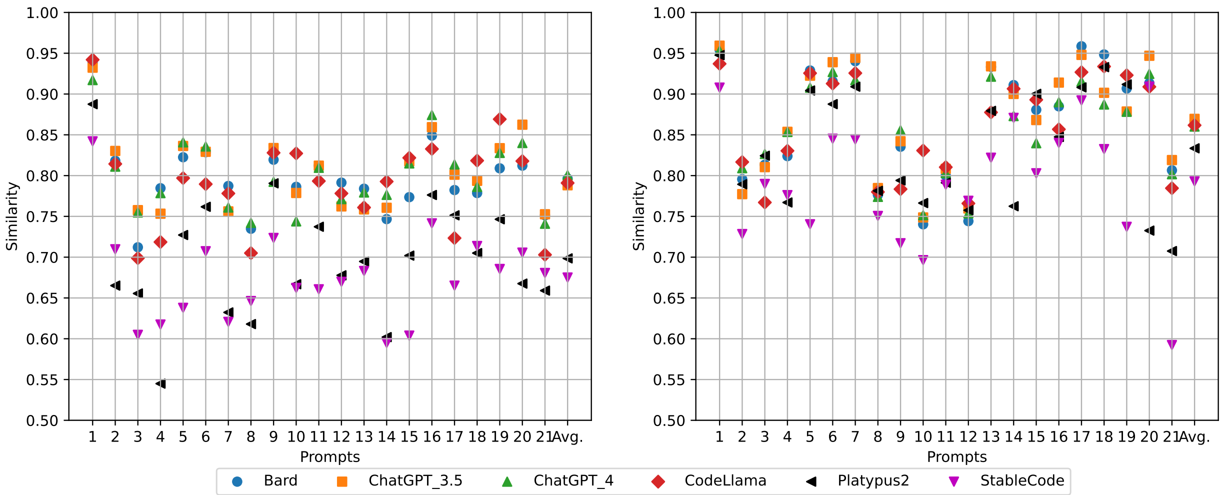

Fig. 4: Average $F3$-Score for each prompt, including average over all prompts (column Avg.), *left* with initial prompt, *right* with prompt optimization.

optimization reduced the required time for these three LLMs, while it increased for Code Llama, Platypus2, and StableCode. The longest and most complex program required a significantly longer execution for most LLMs. This prompt took over 13 minutes for Platypus2 with prompt optimization.

VI. Discussion

Our evaluation reveals that the correctness with regard to our metrics varies depending on the used LLMs. ChatGPT-4 generates the most correct programs, specifically 28 out of 63 and 36 out of 63. Based on the pass@k-score, ChatGPT-4 shows the highest likelihood of producing executable code. In the F_3-Score, Bard outperforms ChatGPT, indicating that Bard-generated programs more closely resemble the reference code. In our study, Platypus2 and StableCode achieve the lowest F_3-score. We discovered syntactic and semantic errors in generated programs, with a focus on the former. Many LLMs struggle with ST syntax, especially in correctly terminating control structures. Semantic errors, particularly involving the improper use of the `RETURN` statement, occur mainly in functions and methods. The computation time of local LLMs is system-dependent, ChatGPT had the highest computation time of the web-based ones. The small size of StableCode was also reflected in its required time. Generating long programs is discouraged, in particular for local LLMs, due to the increased amount of errors and the longer generation times. However, the positive impact of prompt optimization was clearly shown in our study. The additional information significantly improved results, which are reflected in both the F_3 and pass@k scores. Prompt optimization enabled Bard to generate functionally correct programs, although no correct programs were achieved without optimization. Due to the large size of ChatGPT-4, a higher performance regarding ST generation was anticipated. The results furthermore highlight unexpected weaknesses in StableCode and Platypus2. Hardware limitations also impact the performance of locally operated LLMs, as models with fewer parameters and in a quantized form have to be used. By using non-quantized variants, improved results can be expected. Limitations include the prompt collection covering various PLC programming areas but potentially missing certain use cases and technologies. The dataset of 21 prompts might not fully represent the diversity of scenarios. The formulation of prompts may unintentionally favor ChatGPT. Automated solutions for syntax checks and tests of PLC code are not state-of-the-art, presenting challenges for efficient evaluation of generated code. Finally, the replicability of this study is limited due to regular version updates of web-based LLMs. For instance, Bard was replaced by Google's newer LLM Gemini. The performance of updated models regarding ST generation cannot be predicted.

VII. Conclusion

This study investigated among five current LLMs which one is best suited for generating ST. The devised evaluation method applies prompt engineering for typical PLC programming tasks in four iterations and compares the selected LLMs based on the average response time, pass@k, the F_3 score, and CodeBertScore. Our experimental results indicate that, at the time of the study, ChatGPT-4 had the highest capability for generating ST. However, the investigations revealed that current LLMs often produce syntactical errors in generating ST, highlighting significant developmental needs. Considering the development progress in recent years, the performance may continue to improve. Identified limitations have to be addressed in further research, which can explore additional versions of LLMs (e.g., new releases, improved hardware to execute non-quantized models), a larger set of prompts, or refined prompt optimization. The current study was further limited to ST, but other PLC programming languages of IEC 61131-3, including graphical ones, are yet to be explored. Finally, research on fine-tuning LLMs for PLC code generation may allow them to enhance their domain-specific capabilities.

Authorized licensed use limited to: Universite de Rennes 1. Downloaded on March 08,2025 at 09:32:34 UTC from IEEE Xplore. Restrictions apply.

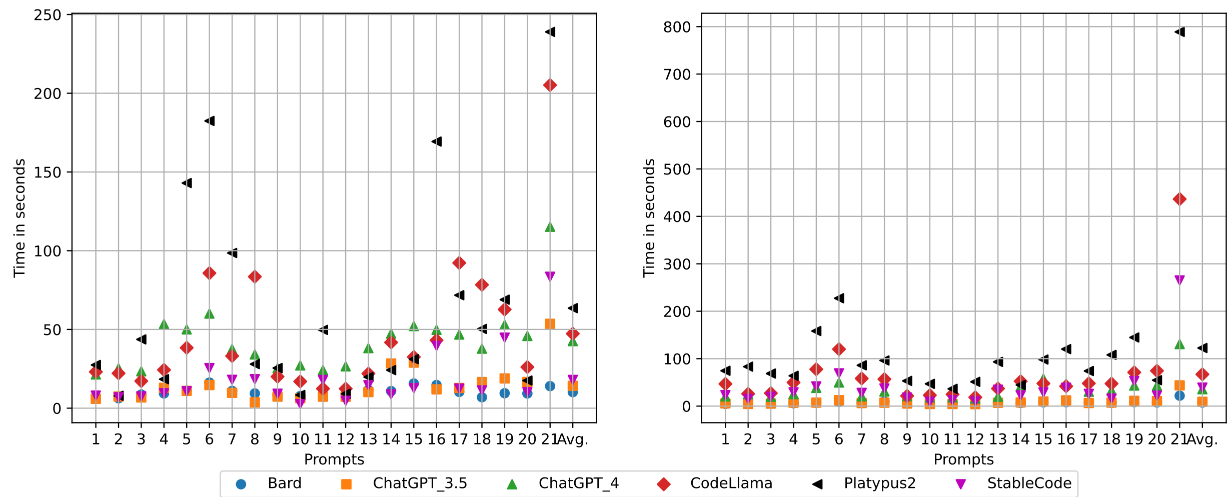

Fig. 5: Average time for generating ST code for each prompt, including average over all prompts (column Avg.), *left* with initial prompt, *right* with prompt optimization.

This direction requires access to the PLC code. In the future, PLC programmers can augment the devised set of instructions with their own prompts.

References

[1] W. Bolton, "Programmable logic controllers," *Newnes*, 2009.

[2] B. Vogel-Heuser, C. Diedrich, A. Fay, S. Jeschke, S. Kowalewski, M. Wollschlaeger, and P. Göhner, "Challenges for software engineering in automation," *Journal of Software Engineering and Applications*, vol. 07, no. 05, pp. 440–451, 2014.

[3] B. Wiesmayr, A. Zoitl, and R. Rabiser, "Assessing the usefulness of a visual programming IDE for large-scale automation software," *Software and Systems Modeling*, vol. 22, no. 5, pp. 1619–1643, 2023.

[4] "ISO/IEC 25010:2023 Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Product quality model," 2011.

[5] P. Vaithilingam, T. Zhang, and E. L. Glassman, "Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models," in *Chi conference on human factors in computing systems extended abstracts*, 2022.

[6] M. T. White, *Mastering PLC Programming*. Packt Publishing, 2023.

[7] N. Nguyen and S. Nadi, "An empirical evaluation of GitHub copilot's code suggestions," in *19th Int. Conf. on Mining Software Repositories*, MSR '22, ACM, 2022.

[8] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin, *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[9] H. Koziolek, A. Burger, M. Platenius-Mohr, and R. Jetley, "A classification framework for automated control code generation in industrial automation," *Journal of Systems and Software*, vol. 166, 2020.

[10] R. Julius, M. Schürenberg, F. Schumacher, and A. Fay, "Transformation of GRAFCET to PLC code including hierarchical structures," *Control Engineering Practice*, vol. 64, pp. 173–194, 2017.

[11] K. Sacha, "Automatic Code Generation for PLC Controllers," in *24th Int. Conf. on Computer Safety, Reliability, and Security*, vol. 3688 of *Lecture Notes in Computer Science*, pp. 303–316, Springer, 2005.

[12] B. Vogel-Heuser, D. Witsch, and U. Katzke, "Automatic code generation from a UML model to IEC 61131-3 and system configuration tools," in *2005 International Conference on Control and Automation*, vol. 2, pp. 1034–1039 Vol. 2, 2005.

[13] M. U. Hadi, Q. Al-Tashi, R. Qureshi, A. Shah, A. Muneer, M. Irfan, A. Zafar, M. B. Shaikh, N. Akhtar, M. A. Al-Garadi, J. Wu, and S. Mirjalili, "Large Language Models: A Comprehensive Survey of its Applications, Challenges, Limitations, and Future Prospects," *preprint*, 2023.

[14] A. Buscemi, "A comparative study of code generation using chatgpt 3.5 across 10 programming languages," *arXiv preprint arXiv:2308.04477*, 2023.

[15] H. Koziolek, S. Gruener, and V. Ashiwal, "ChatGPT for PLC/DCS control logic generation," in *IEEE 28th International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2023.

[16] H. Koziolek, S. Grüner, R. Hark, V. Ashiwal, S. Linsbauer, and N. Eskandani, "LLM-based and retrieval-augmented control code generation," in *IEEE/ACM 46th Int. Conf. on Software Engineering: Companion Proceedings*, 2024.

[17] A. Boudribila, M.-A. Chadi, A. Tajer, and Z. Boulghasoul, "Large language models and adversarial reinforcement learning to automate PLCs programming: A preliminary investigation," in *2023 9th Int. Conf. on Control, Decision and Information Technologies (CoDIT)*, pp. 650–655, IEEE, 2023.

[18] Samarpit Nasa, "Hardware requirements for large language model (LLM) training." https://www.appypie.com/blog/hardware-requirements-for-llm-training, 2023. Accessed: 3.1.2024.

[19] Huggingface, "Quantization." https://huggingface.co/docs/optimum/concept_guides/quantization. Accessed:04.01.2024.

[20] X. Du, M. Liu, K. Wang, H. Wang, J. Liu, Y. Chen, J. Feng, C. Sha, X. Peng, and Y. Lou, "Classeval: A manually-crafted benchmark for evaluating LLMs on class-level code generation," *arXiv preprint arXiv:2308.01861*, 2023.

[21] L. Sonnleithner, B. Wiesmayr, A. Gutiérrez, R. Rabiser, and A. Zoitl, "Complexity of Structured Text in IEC 61499 Function Blocks: A Survey," in *28th Int. Conf. on Emerging Technologies and Factory Automation (ETFA)*, 2023.

[22] S. Zhou, U. Alon, S. Agarwal, and G. Neubig, "CodeBERTScore: Evaluating code generation with pretrained models of code," in *2023 Conference on Empirical Methods in Natural Language Processing*, pp. 13921–13937, ACM, 2023.

[23] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," in *Findings of the ACL: EMNLP*, pp. 1536–1547, 2020.

[24] K. P. Murphy, *Probabilistic Machine Learning: An Introduction*. MIT press, 2022.

[25] Z. A. Wang, "HumanEval: Decoding the llm benchmark for code generation." https://deepgram.com/learn/humaneval-llm-benchmark, 2023. Accessed: 20.2.2023.

[26] Codesys Group, "Codesys." https://de.codesys.com/, 2023. Accessed: 15.1.2024.

[27] R. Maragheh, C. Fang, C. Irugu, P. Parikh, J. Cho, J. Xu, S. Sukumar, M. Patel, E. Korpeoglu, S. Kumar, and K. Achan, "LLM-TAKE: Theme-aware keyword extraction using large language models," in *IEEE Int. Conf. on Big Data*, pp. 4318–4324, 2023.

Authorized licensed use limited to: Universite de Rennes 1. Downloaded on March 08,2025 at 09:32:34 UTC from IEEE Xplore. Restrictions apply.