AutoPLC: Generating Vendor-Aware Structured Text for Programmable Logic Controllers

Donghao Yang[a], Aolang Wu[a], Tianyi Zhang[a], Li Zhang, Xiaoli Lian[b], Fang Liu[b], Yuming Ren, Jiaji Tian
SKLSDE, Beihang University, China
Email: {yangdonghao,aolangwoo,tianyiz,lily, lianxiaoli,fangliu,yumingren,tianjiaji}@buaa.edu.cn

Xiaoyin Che
Siemens AG
Email: xiaoyin.che@siemens.com

[a]Equal contribution, [b]Co-corresponding authors

Abstract—Among the programming languages for Programmable Logic Controllers (PLCs), Structured Text (ST) is widely adopted for industrial automation due to its expressiveness and flexibility. However, major vendors implement ST with proprietary extensions and hardware-specific libraries - Siemens' SCL and CODESYS' ST each differ in syntax and functionality. This fragmentation forces engineers to relearn implementation details across platforms, creating substantial productivity barriers. To address this challenge, we developed AutoPLC, a framework capable of automatically generating vendor-aware ST code directly from natural language requirements. Our solution begins by building two essential knowledge sources tailored to each vendor's specifications: a structured API library containing platform-exclusive functions, and an annotated case database that captures real-world implementation experience. Building on these foundations, we created a four-stage generation process that combines step-wise planning (enhanced with a lightweight natural language state machine support for control logic), contextual case retrieval using LLM-based reranking, API recommendation guided by industrial data, and dynamic validation through direct interaction with vendor IDEs. Implemented for Siemens TIA Portal and the CODESYS platform, AutoPLC achieves 90%+ compilation success on our 914-task benchmark (covering general-purpose and process control functions), outperforming all selected baselines, at an average cost of only $0.13 per task. Experienced PLC engineers positively assessed the practical utility of the generated code, including cases that failed compilation.

Index Terms—Code Generation, PLCs, Large Language Models, Structured Text, IEC 61131-3

I. INTRODUCTION

Programmable Logic Controllers (PLCs) are the cornerstone of Industrial Control Systems (ICSs) within the broader field of Industrial Automation. They play a pivotal role in the operation and management of critical infrastructure across diverse sectors such as energy, manufacturing, and transportation. The significance of PLCs is further evidenced by their growing market projection, with forecasts anticipating a rise to USD 12.20 billion by 2024 and USD 15.12 billion by 2029, marking a robust Compound Annual Growth Rate (CAGR) of 4.37% from 2024 to 2029 [19, 48].

PLCs are programmed predominantly using languages defined by the IEC 61131-3 standard [12]. Among these, Structured Text (ST) is the only high-level, block-structured textual language, with syntax resembling Pascal and C. Due to its high-level features, engineers often use ST to implement data processing, communication protocols, and complex control algorithms. In scenarios where graphical languages like Function Block Diagram fall short in expressiveness, ST becomes the preferred choice [50].

As industrial control systems become increasingly complex, interoperability among heterogeneous devices from different vendors becomes essential[36]. However, vendors typically implement ST with proprietary extensions to IEC 61131-3, including custom syntax, specialized functions, and firmware-specific execution semantics. Notable examples include Siemens' SCL, Rockwell's ST, and CODESYS' ST, each featuring distinct syntax and functionality. This vendor-specific fragmentation imposes a significant learning burden on engineers and highlights the critical need for automated, vendor-aware code generation in industrial control development.

Large Language Models (LLMs) have made remarkable progress in code generation, though primarily for mainstream languages like Java and Python [3, 22, 51, 52]. This focus stems from the abundance of high-quality public code available on platforms like GitHub. However, applying LLMs to ST generation introduces unique challenges. High-quality public ST datasets are scarce, hindering effective training and domain adaptation. Directly transferring frameworks from general-purpose languages often leads to code hallucination [49][16], such as generating Pascal-style control structures. Additionally, LLMs often struggle with vendor-specific functions, particularly those introduced in recent firmware versions, resulting in type mismatches and unsupported function calls.

Despite these challenges, both academia and industry have begun exploring LLM-based automation for ST programming. For instance, LLM4PLC [11] fine-tuned LLMs on limited public dataset, achieving preliminary ST code generation. Koziolek et al. [28] proposed retrieval-augmented generation (RAG) to integrate relevant function blocks into the output, while Agents4PLC [34] introduced a multi-agent framework combining RAG with a feedback-refinement loop to improve accuracy. However, these pioneering explorations generally lack practicality in current industrial environments. The fundamental reasons are twofold:

- **Limited utilization of vendor knowledge.** Accurate ST

code generation requires a precise understanding of vendor-specific syntax and fundamental functions. However, these resources are typically fragmented across technical documentation, lacking structured formats compatible with RAG systems. Although Agents4PLC [34] attempts to curate reusable code snippets and domain knowledge, its opaque knowledge base construction methodology prevents effective reuse. In addition, existing approaches [28, 34] retrieve vendor functions via embedding similarity with control narratives, failing to align high-level requirements and low-level function semantics, resulting in irrelevant retrievals.

- **Lack of interaction with industrial programming environments.** Most approaches use open-source compilers (*e.g.*, MATIEC [8]) without interacting with actual PLC programming platforms [11, 34]. Moreover, existing evaluation benchmarks are small in scale. Fakih et al. [11] evaluate their approach on a small subset of the OSCAT IEC 61131-3 library (i.e., 40 samples) [40], while Haag et al. [16] use the generic APPS Python benchmark, translating it into ST via GPT-4, which may not reflect real-world PLC patterns. Liu et al. [34] develop a custom benchmark tailored to ST-specific features, but only 23 programming tasks are publicly released. These limitations make it difficult to reliably assess the automation potential in real-world industrial scenarios.

To address these challenges, we propose *AutoPLC*, a novel framework for generating vendor-specific ST code. Our approach combines two specialized knowledge bases: (1) *Rq2ST* - a case library of requirement-plan-code triplets enriched with vendor APIs, and (2) *APILib* - a comprehensive function library integrating both IEC 61131-3 standards and vendor extensions. The framework implements an intelligent four-stage workflow: planning (using lightweight NL-defined finite state machines for control logic), case retrieving, API recommendation (combining align their semantics with the intent of each planned step, I/O type matching, and case-based reasoning), and an iterative generate-improve process that actively interfaces with vendor platforms.

To evaluate the effectiveness of *AutoPLC* in generating ST code, we constructed a benchmark dataset of 914 tasks based on open-source resources. The dataset covers two representative ST programming scenarios: 1) General-purpose tasks, such as date processing, string manipulation, file I/O, and numerical computation; 2) Process control tasks, such as robotic arms and conveyor systems. It includes programming tasks for two vendors: the open-source CODESYS' ST language and Siemens' industrial Structured Control Language (SCL) dialect [6].

We implement AutoPLC and conduct experiments on two mainstream vendor platforms, Siemens' TIA Portal [46] and 3S-Smart's CODESYS [6]. Results demonstrate that Auto-PLC outperforms state-of-the-art ST generation methods and general-purpose code models, achieving over 90% compilation pass rate on generating the two ST variants, with the average cost of $0.13 per task. In comparison, the advanced general model Claude-3.5 only achieves 27% on TIA Portal and 58%

on CODESYS. On the Agents4PLC benchmark with 23 tasks, AutoPLC achieves a formal verification pass rate of 78%. To further assess its practical utility, we invite domain experts to review the generated code. The experts provide positive feedback on its effectiveness and applicability in real-world industrial scenarios.

In summary, our main contributions are as follows.

- **Methodology:** We propose the *AutoPLC* framework, which includes a vendor knowledge base construction approach and a LLM-based pipeline for generating ST code from natural language requirements.
- **Benchmark:** We construct a benchmark dataset of 914 requirement–code pairs, covering general-purpose and process control tasks in both CODESYS' ST and Siemens' SCL.
- **Evaluation:** We conduct a comprehensive evaluation of *AutoPLC*, and the results demonstrate that it outperforms state-of-the-art methods, exhibits promising generalizability across ST variants.
- **Open-source Package:** We release a replication package containing the benchmark, source code, and experimental results, available at https://github.com/cangkui/AutoPLC.

II. BACKGROUND AND PROBLEM STATEMENT

A. Background

A PLC project typically consists of multiple units, such as FUNCTION_BLOCKs with internal state and stateless FUNCTIONs, each encapsulating specific control logic and interacting with one another through defined input/output interfaces, covering both general-purpose logic (e.g., arithmetic, comparison, timing) and device-specific control (e.g., motors, sensors, actuators).

Fig.1a demonstrates an actuator control logic implementation using a finite state machine in CODESYS' ST. The solution maintains internal state variables and updates outputs according to system timing constraints. Figs. 1b-1c compare the syntactic differences when implementing a FIFO (First-In-First-Out) FUNCTION_BLOCK across two vendor platforms:

- Fig.1b shows the implementation for Beckhoff TwinCAT-based PLCs, which support object-oriented programming. Developers can use `POINTER` types and `ANY` generics to handle arbitrary data types, and encapsulate logic in methods such as `Enqueue`, offering both high-level abstraction and low-level memory control similar to general-purpose languages.
- Fig.1c presents the counterpart for Siemens S7-1200/1500 PLCs, which adopt a procedural style. It relies on proprietary data type `Variant` to handle arbitrary data, and uses dedicated instructions like `MOVE_BLK_VARIANT` to manipulate this type.

Clearly, developing units for PLC project requires a thorough understanding of vendor ST's syntax, libraries, and programming paradigms. This increases the complexity of cross-vendor development.
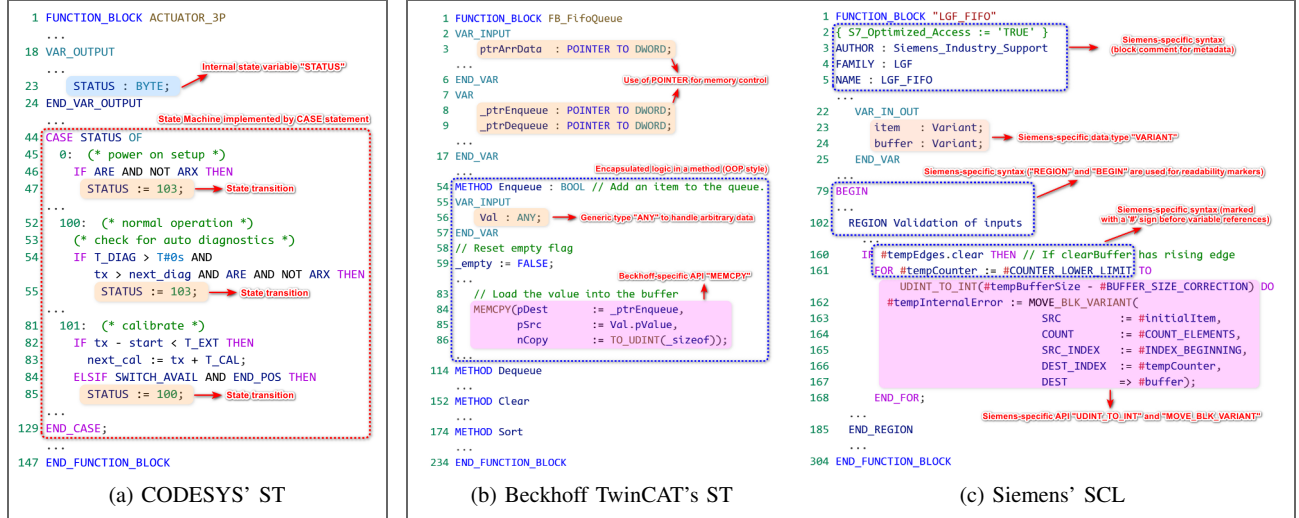
Fig. 1: Example code segments implemented by different vendors. (a) is ACTUATOR, (b) and (c) are both FIFO.

B. Problem Statement

Given $Task = \langle Req, IO, Type, Vendor\_Target \rangle$ as task description, where Req is a natural language description of the desired functionality, $IO = \{(t_i, v_i, s_i, d_i)\}_{i=1}^{n}$ is a list of I/O variables with type t_i, name v_i, description s_i, and direction $d_i \in \{IN, OUT, INOUT\}$, Type specifies whether to generate a FUNCTION_BLOCK or a FUNCTION, and Vendor_Target specifies the target PLC platform. The goal is to generate a set $\mathcal{C} = \{c_1, c_2, \ldots, c_k\}$ of vendor-aware ST units such that each c_i is expected to be compilable on the specified platform or require only minimal manual adjustment.

III. APPROACH: AUTOPLC

AutoPLC provides an end-to-end solution for generating industrial ST code from natural language requirements. The framework comprises two fundamental components: a vendor-specific knowledge infrastructure and a four-stage LLM-based workflow, as shown in Fig.2.

AutoPLC operates through two specialized knowledge bases. *APILib* serves as a comprehensive repository of vendor fundamental APIs, complete with functional descriptions and usage examples, while *Rq2ST* maintains a collection of proven requirement-to-code implementations. The generation proceeds through four key stages: Planning, Case Retrieving from *Rq2ST*, API Recommendation from *APILib*, and final Code Generation based on interaction with platforms.

Thus, extending *AutoPLC* to new ST variants is straightforward - simply update the underlying *APILib* and *Rq2ST* components, and integrate the compilation service of target PLC vendor platform during the synthesis phase. This modular approach preserves the core generation workflow while adapting to different vendor environments.

A. Vendor-specific Knowledge Base Construction

APILib assists correct usage of vendor-specific functions(APIs), while *Rq2ST* enables *AutoPLC* to learn from proven implementation experience.

APILib provides structured metadata for the fundamental APIs (usually hardware-aware), systematically collected from vendor documentation. Each API entry includes complete specifications (name, description, parameters) and practical examples. To support API Recommendation in Stage 3, we use GPT-4o to generate a three-dimensional index for each API, comprising: (1) a functional summary, (2) typical usage scenarios, and (3) ST-specific keywords. This index serves as a compact semantic representation for efficient retrieval. Formally, each API is defined as $API = \langle Name, Desc, Params, Examples, Index \rangle$, where $\text{Index} = \langle S_{summary}, S_{scenario}, K_{keywords} \rangle$ contains the generated summary, scenarios, and keyword set.

Our successful case library *Rq2ST* draws from multiple sources, including vendor-specific reusable function sets (like Siemens' SCL libraries or CODESYS' instruction set) and validated industrial code repositories. Each case is represented as a quadruple $\langle Task, Plan, Code, APIs \rangle$ containing the task description, the design plan, the implementation, and the fundamental APIs. While $Task$ and $Code$ are obtained directly, the high-level $Plan$ is derived by GPT-4o through code analysis, and the $APIs$ are identified by matching the ST code against *APILib*.

B. Stage 1: Planning

Planning prior to code implementation serves as a critical translation mechanism, converting high-level natural language task descriptions into executable **step-by-step** specifications. It has been empirically validated as an effective approach for bridging abstract requirements with concrete implementations, ultimately enhancing the quality of generated code [10, 21].

State-of-the-art planning for code generation is typically implemented using advanced LLMs [10, 20, 21], and some studies propose pseudocode-based planning to enhance code generation accuracy [30], we identify two critical limitations in applying these techniques directly to our task:
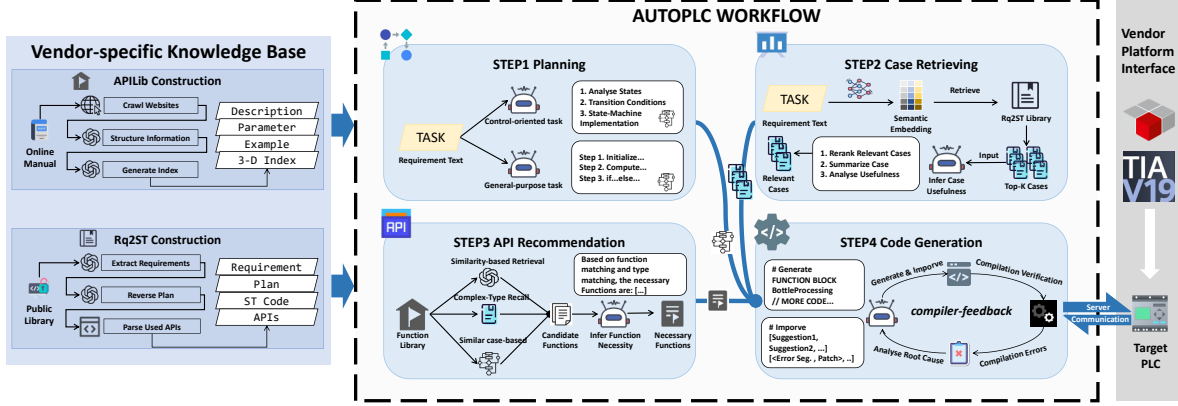
Fig. 2: Framework of AutoPLC.

- *Overly generic plans fail to capture control logic specifics.* Public LLMs, trained predominantly on general-purpose tasks, tend to produce plans that inadequately represent the state transitions fundamental to process control tasks.
- *Pseudocode-formatted plans introduce syntactic noise in ST generation.* According to our observation, when plans are expressed in pseudocode, the code-generation LLM may directly replicate pseudocode constructs due to its inability to properly distinguish them from valid ST syntax.

Our approach maintains the use of advanced LLMs for planning while introducing key adaptations. First, we implement task-type classification (process-control vs. general-purpose) through targeted prompting. For process-control tasks, our Planner directs the LLM to: (1) identify system states and their transitions, and (2) formulate a natural-language state machine representation, with execution planning reduced to state machine traversal. For general-purpose tasks, we follow conventional linear planning methodologies [30, 53] that outline procedural logic through sequential computational steps.

We deliberately employ natural-language plans instead of pseudocode to avoid syntactic contamination. This abstraction preserves the LLM's flexibility in selecting appropriate ST implementations while preventing direct syntax carryover from intermediate planning representations.

C. Stage 2: Case Retrieving

Despite the structured task decomposition from the planning stage, generating compliant ST code remains non-trivial due to the dialect variations across vendor implementations and the fact that ST is under-represented in the pre-training corpora of LLMs. While domain adaptation through fine-tuning represents a common approach for LLM specialization [14, 15, 38], the per-vendor cost of continuously curating large-scale ST corpora quickly outweighs the benefits, making fine-tuning economically unattractive for our multi-vendor setting. Recent advances in RAG [29, 55] offer a more viable solution by capitalizing on LLMs' demonstrated in-context learning capabilities [5, 9] : the compact, high-quality cases in Req2ST are

fetched on demand, letting the LLM learn in context without ever retraining.

Our retrieval process begins by selecting 5 candidate cases based on embedding similarity between the task description and case's metadata, leveraging Zhipu AI's retrieval service[1]. Next, we employ an LLM to rerank these candidates according to their potential contribution to the current task, ultimately choosing the three most beneficial examples for final inclusion. We then dynamically inject their requirements, plans, and ST code as few-shot examples in Stage 4's structured prompt. Although these retrieved cases inherently include API usage information, we do not directly inject these APIs as few-shot demonstrations. Instead, the relevant APIs undergo processing and refinement in Stage 3 before being systematically integrated into the generation prompt.

D. Stage 3: API Recommendation

The goal of this stage is to recommend potentially useful APIs for the ST generation. This approach has proven effective in improving the performance of code-generation LLMs, particularly for less familiar languages [31]. The challenge of API usage is further compounded by vendor-specific extensions to the standard (e.g. Siemens adds hundreds of proprietary low-level library functions, named Extended Instructions). Consequently, general-purpose LLMs often fail to accurately identify the appropriate hardware-aware functions across diverse scenarios, especially given the concrete dialect and limited resources of ST.

We adopt a two-step retrieval and filtering strategy to balance recall and precision. Recall is prioritized, as providing a broad candidate set enables the LLM to make informed selections. Without sufficient recall, the LLM lacks viable options, even if it can assess relevance accurately [32]. To further enhance recall, we collect candidate APIs from three dimensions during retrieval: similarity-based retrieval, parameter-based augmentation, and similar-case-based augmentation. Denote the APIs collected from these dimensions as $\Lambda_{similarity}$, Λ_{par},

---

[1]https://bigmodel.cn/dev/howuse/retrieval

and $\Lambda_{simCase}$. The final API set for this stage is the union of these three sets.

The set $\Lambda_{similarity}$ is obtained by using each step in the planning process from Stage 1 as a query to retrieve candidate APIs, calculating relevance against a 3-dimensional index of each API. We employ Okapi BM25 [42], a method widely used in RAG for code-generation tasks [33, 54]. The set Λ_{par} includes potential APIs that share input or output types with the ongoing task, focusing specifically on ST-specific complex types (e.g., structured data types like DTL). The set $\Lambda_{simCase}$ is constructed by retrieving APIs from the top three relevant cases identified in Stage 2.

Next, we apply LLM-based filtering to refine the final set. The filtering needs to consider both functional relevance (whether the API's purpose matches the task) and interface compatibility (whether parameter types align), which cannot be fully determined through keyword matching alone. Thus, we prompt an LLM to evaluate and select the most suitable functions. To maintain prompt efficiency, we package the indexed fields of up to 15 candidate functions into each prompt. While no filtering is perfect, research shows that LLMs can effectively assess the suitability of retrieved content, making this approach both practical and empirically validated [32, 44].

E. Stage 4: Code Generation

In this stage, we dynamically construct a structured prompt by combining the task description with contextual elements: step-by-step plan, potentially useful APIs, and requirement-plan-code few-shot examples. This rich prompt enables the LLM to generate high-quality ST code. To ensure syntactic correctness, we implement a feedback-driven modification loop that iteratively improves initially generated code using compiler error analysis (called the Self-improvement mechanism).

While open-source ST validation tools like MATIEC [8] and IECChecker [24] exist, they lack support for the diverse and evolving ST dialects. Our solution integrates vendor platform's services, e.g., TIA Openness (Siemens TIA Portal), providing access to authentic compilation diagnostics directly within our generation workflow.

When fixing errors in the generated code, we follow a practical way informed by engineers experience: we tackle *declaration errors* first before moving to *implementation errors*, since unresolved declarations tend to create a flood of cascading issues (e.g. "undefined identifier"). Our repair process works by having the LLM carefully inspect each error, figure out potential fixes, and then produce snippet-replacement patches. These targeted patches get applied to the problematic snippets, allowing for precise corrections without having to regenerate the entire code. We keep refining this way until the code compiles cleanly or we hit our iteration limit.

IV. EVALUATION

A. Research Questions

RQ1 (Effectiveness Evaluation): What is the effectiveness of *AutoPLC* in generating ST across vendors?

RQ2 (Ablation Study): How do AutoPLC's Planning, Case Retrieving, API Recommendation, and the Self-improvement mechanism for code generation affect overall performance?

RQ3 (Manual Evaluation): How do developers perceive the quality of code generated by *AutoPLC*?

B. Datasets and Metrics

1) Datasets: Due to the sensitivity of industrial applications, publicly available ST code is scarce. Existing studies, such as Fakih et al. [11], constructed benchmarks based on public ST libraries but did not release their datasets, limiting transparency and reproducibility. In this study, we build our benchmark datasets by carefully curating 914 cases from three authoritative sources representing two major industrial vendors' ST variants (CODESYS' ST and Siemens' SCL):

1) CODESYS Implementation of OSCAT Library (718 cases) [40]: A widely-adopted, community-maintained open-source library containing general ST functions, network utilities, and building automation components.
2) Siemens LGF Library (151 cases) [45]: An industrial-standard library optimized for Siemens automation systems, offering comprehensive general-purpose functions.
3) Siemens Competition Dataset (45 cases) [4]: Real-world industrial programming tasks from Siemens featuring process control task with practical constraints.

For each function in these library documentations, we extract the description and input/output parameters as task description, and the corresponding code implementation as the ground truth. We further analyze the code complexity based on the vendor's official implementation. Table I provides detailed statistics including case counts, lines of code (LOC), and input/output variables.

Additionally, we evaluate AutoPLC on the Agents4PLC Benchmark [34], which includes 23 PLC programming tasks spanning diverse scenarios.

TABLE I: Statistics of Our Benchmark.

Dataset	#Cases	Lines of Code		Input/Output Variables	
		Average	Median	Average	Median
OSCAT	718	34.40	20	4.39	2
LGF	151	91.46	69	5.50	4
Competition	45	84.68	76	6.12	6

2) Metrics: Due to the lack of standardized ST benchmarks [11], we adopt dataset-specific evaluation metrics. For our constructed datasets (i.e., LGF, Competition, and OSCAT), we report the *Compilation Pass Rate (Pass Rate)* and *Average Error Count (Avg. Errors)*, based on actual compilation results from CODESYS and Siemens TIA Portal. We omit formal verification due to unavailable public specifications and prohibitive construction costs for large-scale datasets [37]. Following Koziolek et al. [27]'s OSCAT framework, we implement rigorous validation by first verifying test cases against the ground truth implementations (retaining only tasks with ≥ 5 passing cases), then manually executing test suites in CODESYS with full success required for verification.

For the 23 tasks in the Agents4PLC benchmark, we follow its official protocol [34], using `rusty` for syntax checking and `plcverif` for formal verification.

C. Implementations of AutoPLC

Our implementation generates ST code for two major industrial platforms.

- *Siemens' SCL*: We construct the Rq2ST knowledge base using the Competition and LGF datasets (Section IV-B1), augmented with an APILib derived from Siemens' official documentation[2]. During feedback-based code generation (Stage 4), the system interfaces with TIA Portal V19 on S7-1200/1500 PLCs (firmware v2.9) to validate compilation.
- *CODESYS' ST*: The Rq2ST knowledge base integrates the OSCAT dataset and APILib from CODESYS' official documentation. Compilation is performed using CODESYS V3.5 SP20 Patch 5.

To prevent data leakage in code generation (Stage 2 Case Retrieving), we implement a strict filtering mechanism: for each task t, any case e is excluded from retrieval (i.e., omitted from *Rq2ST*) if their textual names exhibit mutual substring containment (case-insensitive). While some cases implementing similar logic to the task may remain, we argue that this reflects real-world project conditions—and is precisely why RAG is beneficial.

In Stage 4, we set the maximum iteration count to 3. For the backbone LLM, we selected Claude-3.5-Sonnet [2] due to its robust code generation capabilities. Our evaluation shows that *AutoPLC* achieves practical efficiency, with a cost of $0.13 per task (matching LLM4PLC) and a processing time of 52 seconds, making it viable for real-world deployment.

D. Baselines

We compare AutoPLC against state-of-the-art baselines, including LLM4PLC [11], Agents4PLC [34], MapCoder [20], and several widely-used general LLMs.

For LLM4PLC, we adapt the framework to a fully autonomous setting by removing the manual interventions while retaining its iterative feedback from MATIEC [8](syntax checking) checking and NuXMV [23](formal verification) , following its original SMV-based verification strategy. Since the original paper does not specify iteration counts, we standardize this to three iterations, consistent with AutoPLC. To ensure fairness, we adopt the latest GPT-4o as the backbone, as GPT-4 was reported but not version-specified in the original paper [11].

Due to the lack of the released implementation details for Agents4PLC, we are unable to reproduce its results, but we include a comparison against the reported performance in their paper on their released benchmark (Agents4PLC Benchmark) [34]. For MapCoder, we use the source code downloaded from the official GitHub directly, and set its internal `self.LANGUAGE` configuration to ST/SCL, while keeping the other settings unchanged. We employ the same

backbone LLM (Claude-3.5-Sonnet) as used in our *AutoPLC* system.

Additionally, we include six general LLMs: DeepSeek-Chat [15], Qwen2.5-Coder-Instruct (7B) [18], GPT-4o [1], GLM-4-Plus [13], Claude-3.5-Sonnet [2], and Llama-3.1-Instruct (8B) [35]. These models are widely used as coding assistants and cover a diverse range of architectures and capabilities.[3] To ensure consistency, all models are evaluated using the same prompt templates, which are provided in the supplementary replication package.

V. RESULTS AND ANALYSIS

A. Addressing RQ1: Effectiveness Evaluation

Table II presents the comparative results of *AutoPLC* against all baselines across four datasets. *AutoPLC* demonstrates consistent superiority over all baselines in both evaluation metrics, achieving compilation pass rates exceeding 90% while maintaining fewer than 2 average errors across all datasets.

More observations can be made as follows.

- *Claude-3.5-Sonnet shows competitive performance* among all baselines, rivaling specialized PLC LLMs like LLM4PLC and MapCoder. As our approach's backbone, it ranks second-best across our three datasets. Nevertheless, *AutoPLC* achieves remarkable improvements over Claude-3.5-Sonnet: pass rates increase by 60.95% (OS-CAT), 508.8% (LGF), and 241.6% (Competition), while average errors decrease by 72.6%, 88.1%, and 80.6% respectively. On the Agents4PLC benchmark, *AutoPLC* achieves 100% compilation success and the highest validation pass rate, outperforming LLM4PLC and MapCoder by 1620.8% and 38.3% respectively. Notably, while MapCoder also uses Claude-3.5-Sonnet, it underperforms the native model in some cases (e.g., on the LGF dataset, Claude surpasses MapCoder by 26.1% in compilation pass rate and produces 35.1% fewer errors). This contradicts the expected performance gains from multi-agent systems in general NLP tasks [20], likely because MapCoder's example-recalling agent depends on model's internal knowledge that proves ineffective for scarce ST examples, precisely where our vendor-specific knowledge bases add value.
- *LLM4PLC underperforms expectations*. Our implementation reveals that the base GPT-4o outperforms LLM4PLC in both ST and SCL generation. Unlike the original LLM4PLC paper, we employ fully automated verification via MATIEC (syntax checking) and NuXMV (formal verification). Our analysis identifies two key limitations: ❶ MATIEC sometimes misflags correct code as erroneous, providing misleading feedback; ❷ GPT-4o struggles to generate valid formal model and specifications for verification. These findings suggest that: (1) industrial vendor platforms should replace open-source MATIEC to improve reliability, and (2) human intervention remains essential for formal verification at this stage.

---

[2]https://docs.tia.siemens.cloud/p/plc-programming-with-simatic-s7

[3]Due to API availability, we use DeepSeek-V3 for the Agents4PLC benchmark and DeepSeek-V2.5 for others.

TABLE II: Comparison results of *AutoPLC* and baselines in generating ST and SCL code.

Baselines	LGF (SCL)		Competition (SCL)		OSCAT (ST)		Agents4PLC (ST)	
	Pass Rate	Avg. Errors	Pass Rate	Avg. Errors	Pass Rate	Avg. Errors	Pass Rate	Valid. Rate
Llama-3.1-Instruct(8B)	0.66%	15.65	0.00%	22.84	5.98%	28.14	4.55%	0.00%
Qwen2.5-Coder-Instruct(7B)	5.96%	10.92	5.61%	9.56	18.08%	13.14	73.91%	17.39%
GLM-4-Plus	6.62%	10.54	6.67%	9.36	35.05%	12.22	82.61%	30.43%
GPT-4o	8.61%	18.76	20.00%	13.49	35.47%	8.39	82.61%	43.48%
DeepSeek-Chat	3.97%	14.53	22.22%	13.27	42.70%	7.68	82.61%	39.13%
Claude-3.5-Sonnet	15.23%	9.60	26.67%	6.53	57.72%	5.76	78.26%	52.17%
LLM4PLC	7.28%	16.54	9.09%	21.80	30.74%	15.52	36.36%	4.55%
Agents4PLC(paper)	-	-	-	-	-	-	100.00%	60.87%
MapCoder	11.26%	12.97	35.56%	18.34	56.96%	8.60	82.61%	56.52%
AutoPLC	**92.72%**	**1.14**	**91.11%**	**1.27**	**92.90%**	**1.58**	**100.00%**	**78.30%**

- *Our approach shows robust performance across two variants, whereas all baselines demonstrate 3.03-10.75× better ST than SCL generation performance (compare the OSCAT and LGF datasets).* Models show highest pass rates on OSCAT, moderate performance on Competition, and poorest results on LGF, a trend correlating with average LOC (see Table I. Longer code usually indicates greater complexity). For error analysis (limited to LGF and Competition due to compiler differences), models consistently make more errors on LGF, aligning with pass rate findings. However, our approach does not show obvious difference between these two SCL datasets due to the usage of two knowledge bases and the feedback-based improvement during code generation. The baselines exhibit a pronounced performance gap between ST and SCL generation, as evidenced by OSCAT (ST benchmark) and LGF (SCL benchmark) evaluations, where ST achieves 3.03-10.75× higher pass rates. This disparity stems from two factors: ❶ OSCAT's alignment with ST's widespread industrial adoption (e.g., Siemens ecosystems) versus LGF's specialization for niche controllers, resulting in richer ST training data; and ❷ SCL's dependence on Siemens-specific extensions beyond IEC 61131-3, which reveal baseline models' knowledge limitations. Notably, *AutoPLC* maintains robust performance across both language-benchmark pairs.

Execution-based evaluation. To evaluate functional correctness beyond compilation, we executed the generated test suites on AutoPLC and MapCoder—the latter being a specialized code-generation model with strong performance on general-language test-case accuracy [20]. Following the methodology in Section IV-B2, we selected the intersection of compilation-passing cases from both approaches. After filtering, we retained 71 tasks, each with approximately 10 test cases.

The results show that AutoPLC achieved full-suite success on 35 tasks, while MapCoder succeeded on 34—both close to 50%. Across all 487 test cases, the two models passed 76%. This validates our approach, especially considering OSCAT's 92.90% compilation pass rate. Notably, MapCoder's low compilation pass rate but high test case pass rate suggests that the primary challenge in ST generation lies in syntax compliance, a prerequisite for test execution. All raw test results are available in the replication package.

Answer to RQ1: *AutoPLC* markedly outperforms state-of-the-art baselines across all datasets, with higher compilation pass rate and lower error counts.

B. Addressing RQ2: Ablation Study

To rigorously evaluate *AutoPLC*'s design, we conduct an ablation study by sequentially disabling Planning, Case Retrieving, API Recommendation, and Self-improvement mechanism in Stage 4. Notably, the vendor-specific knowledge bases (*Rq2ST* and *APILib*) serve as foundational infrastructure that cross-cut multiple Stages - particularly enabling Case Retrieving (Stage 2) and API Recommendation (Stage 3). Therefore, rather than ablating these shared knowledge bases directly, we assess their implicit impact through the ablation of the dependent algorithmic components.

The outcomes are presented in Table III. To quantify the extent of these variations, we include the change ratios accompanied by directional arrows and red figures to signify the degree of reduction or increase, respectively. The following observations can be made:

- *All modules contribute to the generation of ST and SCL code, as evidenced by increased Pass Rates and reduced Average Errors when incorporated.*
- *The impact of the stages is more pronounced in the SCL datasets compared to the ST datasets.* For example, removing the Case Retrieving causes an 8.39% drop in the Pass Rate for OSCAT, but 35.01% and 19.51% for LGF and Competition, respectively, highlighting a 26.62% gap between public libraries. This result reinforces the conjecture that general LLMs lack domain-specific expertise for SCL programming. As shown in Huang et al. (2024) [17], incorporating external knowledge and feedback strategies can substantially boost performance.
- *The case-retrieving and the self-improvement contribute substantially to performance gains.* Removing the case-retrieving reduces the Pass Rate by up to 35.01%, and removing the self-improvement leads to a decrease of up to 29.27% in the Pass Rate. Notably, case-retrieving removal has a more severe impact on the LGF benchmark compared to Competition benchmark, likely because LGF tasks, focused on general-purpose functions, rely heavily on

TABLE III: Ablation study results.

Baselines	LGF (SCL)		Competition (SCL)		OSCAT (ST)		Agents4PLC (ST)	
	Pass Rate	Avg. Errors	Pass Rate	Avg. Errors	Pass Rate	Avg. Errors	Pass Rate	Valid. Rate
w/o Planning	91.39%$_{\downarrow1.43\%}$	2.52$_{\uparrow121.05\%}$	86.67%$_{\downarrow4.87\%}$	1.24$_{\downarrow2.36\%}$	87.13%$_{\downarrow6.22\%}$	2.79$_{\uparrow76.58\%}$	95.65%$_{\downarrow4.35\%}$	69.57%$_{\downarrow11.15\%}$
w/o Case Retrieving	60.26%$_{\downarrow35.01\%}$	5.08$_{\uparrow345.61\%}$	73.33%$_{\downarrow19.51\%}$	3.09$_{\uparrow143.31\%}$	85.10%$_{\downarrow8.39\%}$	1.93$_{\uparrow22.15\%}$	86.96%$_{\downarrow13.04\%}$	43.48%$_{\downarrow44.47\%}$
w/o API Recommendation	84.77%$_{\downarrow8.57\%}$	2.82$_{\uparrow147.37\%}$	84.44%$_{\downarrow8.93\%}$	2.75$_{\uparrow116.54\%}$	87.85%$_{\downarrow5.44\%}$	1.93$_{\uparrow22.15\%}$	100.00%	73.91%$_{\downarrow5.61\%}$
w/o Self-improvement	66.23%$_{\downarrow28.57\%}$	7.02$_{\uparrow515.79\%}$	64.44%$_{\downarrow29.27\%}$	1.57$_{\uparrow23.62\%}$	84.80%$_{\downarrow8.75\%}$	2.89$_{\uparrow82.91\%}$	100.00%	69.57%$_{\downarrow11.15\%}$
AutoPLC (Full)	**92.72%**	**1.14**	**91.11%**	**1.27**	**92.90%**	**1.58**	**100.00%**	**78.30%**

Siemens' SCL extensions. For the Agents4PLC benchmark, excluding the case-retrieving critically degrades both Compilation Pass Rate and Validation Satisfied Rate, underscoring its role in supplying proven, domain-appropriate logic experience that help raise the overall reliability of control programs.

- *The API-recommendation and the Planning stages provide complementary improvements and are still valuable.* Although their individual impact is less dramatic than Case Retrieving and the Self-improvement, they contribute to steady performance gains across datasets, validating their role in assisting function selection and decompose requirement.

Answer to RQ2: The Case Retrieving and the Self-improvement drive notable gains, while the API Recommendation and the Planning provide complementary support.

C. Addressing RQ3: Manual Evaluation

Till now, we have assessed quantitative measures by evaluating the compilation pass rate and the count of errors for each snippet of generated code. However, we aim to delve deeper into understanding *to what extent can the generated code be practically useful in real engineering scenarios?* To explore this question, we have carried out a user study where participants are invited to manually evaluate aspects such as *correctness, conformance to industry coding standards, modifiability, safety, and usefulness* for the generated code.

- Correctness: On a scale from 1 to 5, to what degree the provided code fulfills the specified tasks.
- Conformance to Industry Coding Standards: Rated from 1 to 5, this measures how well our code aligns with industry practices and norms, including aspects like variable naming conventions, and structuring of branches and loops.
- Modifiability: On a scale from 1 to 5, how easily our code can be altered or expanded upon, particularly within the ever-changing industrial environment.
- Safety: This category, scored from 1 to 5, explores the extent to which our code could be executed safely within a real-world industrial setting.
- Usefulness: On a scale from 1 to 5, usefulness gauges the extent to which our code serves as a helpful reference for the evaluator's own coding work.

The first four metrics are designed for cases that successfully pass compilation, while the final metric is specifically intended for cases that fail to compile. To guarantee the validity of the evaluations, participants were asked to provide explanations for their assigned scores.

We randomly assembled a collection of 30 cases from each of our datasets, amounting to a total of 90 cases. Each case includes a specified requirement and its associated code segment. In every individual benchmark, 20 cases result in successful compilation, while the remaining 10 lead to compilation failures.

We recruited five experts. Three are veteran automation electrical engineers with extensive PLC development experience, counting 8, 7, and 2.5 years respectively. And two are doctoral candidates with specializations in electrical industrial automation, each possessing three years of ST programming experience applied to diverse assignments and projects with varying degrees of complexity. These participants were engaged through public job posting platforms and had no prior knowledge of the authors. They were each responsible for assessing between 10 to 15 cases based on their time constraints. Ultimately, we collected 65 annotations(due to limited expert availability) : 25 for OSCAT (15 passing/10 failing), 17 for LGF (12 passing/5 failing), and 23 Competition entries (14 passing/9 failing). Figure 3 depicts the distributions of scores, including min, max, median, and mean values for correctness, conformance, modifiability, and safety among passing cases, alongside usefulness scores for failing cases.
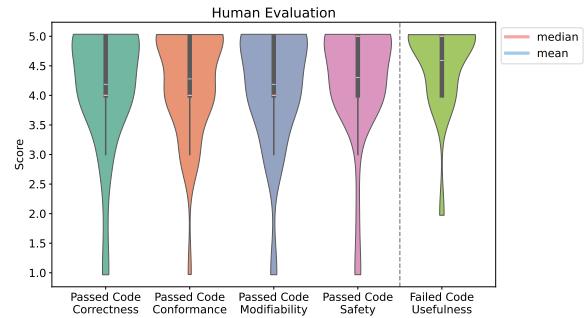

Fig. 3: Violin plot showing the manual evaluation results.

For the *passed samples*, the average ratings for the four concerns across the three datasets are 4.18, 4.28, 4.19, and 4.53, respectively. Notably, the median score for safety reaches the maximum of 5.0. These scores are particularly impressive, especially considering that our participants grounded their assessments in practical applicability. They offered numerous constructive critiques and suggestions for the generated code. For instance, the engineer with seven years of experience, who adheres to a stringent standard for industrial control coding quality, determined that seven out of ten samples satisfied the

stipulated requirements. Nevertheless, he highlighted several issues from various perspectives. For example, he flagged a potential safety concern: "*In industrial environments, the validity of input data must be taken into consideration. An error or default value should be returned when the frequency is zero or negative to prevent runtime errors that could compromise system safety.*" Other participants provided expert feedback on the code as well. One participant identified a Modifiability issue: "*LLMs may not be familiar with the functionality of the Find function. It can return the position within the string by itself; there is no need to use a for loop to search through the string.*" One correctness issue is about the lack of domain knowledge of industrial control: "*In manual mode, the interlock switch between forward and reverse rotation does not reflect the actual situation since the relay action is delayed; therefore, we cannot directly determine the status using the button's state.*"

We discerned that among the 41 cases that passed, four were appraised with *medium correctness*. A closer inspection of the comments revealed that *only one case was deemed incapable of fulfilling the requirements*. The remaining three raised some issues, such as intricate logic or unstable outcomes. Take, for instance, a task within LGF requiring the code that *generates a random value of the real number type within a specified range*. While the participant confirmed the fundamental functional correctness, he also expressed reservations about *the potential for the nanosecond component of system time to be recurrent or erratic in some PLCs due to disparate time resolutions. Consequently, it is recognized that our produced code may not guarantee entirely random values upon each invocation.*

Concerning *conformance*, seven cases each exhibit one issue, with four related to *naming conventions and commentary*, two associated with *unnecessary use of extensive data structures*, and one case highlighting an *improper mix of manual and automated procedures*. Six cases were rated as having medium *modifiability*, with the main points of contention being *ambiguous naming* (three cases), *unclear logical transitions* (two cases), and *improper utilization of process variables*. Lastly, five cases were considered to have medium *safety* concerns, primarily due to *inadequate input verification and subpar handling of exceptions or errors*. We highlight these cases because, in the context of industrial control code, non-functional quality attributes are crucial. Such aspects warrant greater focus and consideration from the community.

Despite containing some errors, the *failed samples* received scores of 4.40, 4.30, and 4.89 within the three datasets, indicating that all participants regarded them as valuable for their coding tasks. Annotations from four participants consistently praised the samples' usefulness. For example, one participant observed, "*In industrial settings, it's common to encounter situations where slope processing is necessary to manage temperature rise and acceleration rates. ... The program also includes an error handling mechanism... While there is room for code Modifiability improvements, its complete and clear functionality means it can still be deployed as-is, offering significant gains in time efficiency.*"

Answer to RQ3: *AutoPLC* generated code that was positively evaluated by experts on selected datasets, though some cases revealed issues such as insufficient input validation or incomplete error handling.

VI. DISCUSSION

A. Error Types Analysis

TABLE IV: Error types analysis of AutoPLC and Claude.

Error Types	OSCAT		LGF		Competition	
	Claude	Ours	Claude	Ours	Claude	Ours
Undefined error	1.84	0.50	5.88	0.77	3.00	0.56
Mismatch error	3.39	0.66	0.97	0.006	1.09	0.02
Call error	0.05	0.004	2.12	0.01	2.00	0.27
Type conversion error	0.25	0.18	0.54	0.25	0.49	0.20
Others	0.21	0.20	0.17	0.10	0.04	0.22

Through systematic analysis of frequent error patterns, we identify current limitations of *AutoPLC* and outline pathways for future enhancement. Specifically, to evaluate the improvements introduced by our proposed components, we systematically categorize error types from both *AutoPLC* and its backbone model, Claude, into five distinct classes based on compiler diagnostics: undefined errors, mismatch errors, call errors, type conversion errors, and other errors. Table IV reports the average error count of each type across benchmarks, with color highlights for visual comparison.

Undefined errors were the most frequent error type in a majority of the benchmarks. While often triggered by referencing undeclared variables, they commonly stem from invalid definitions in the variable declaration section—*e.g.*, illegal initializations—that cause variables to fail to be declared successfully, leading to downstream failures. Our staged repair mechanism mitigates this by prioritizing declaration fixes, reducing undefined error rates by 72.8–86.9% (vs. Claude).

Mismatch errors (e.g., unmatched parentheses, symbols) were also prevalent. These were effectively reduced by retrieving cases from the same ST variant, which avoids structural hallucinations. We observed a reduction of 80.5–99.4% compared to Claude.

Call errors were particularly common in the LGF and Competition benchmarks, and typically stemmed from incorrect instruction usage—such as referencing non-existent instructions or mismatching parameters. These errors reflect limited model familiarity with vendor-specific instructions. AutoPLC mitigates this via instruction recommendation and full signature integration.

Type conversion errors arise when argument types are not properly cast or when invalid conversion syntax is used—e.g., `TO_REAL` instead of `REAL_TO_REAL`. These are frequent in ST due to its stricter type constraints. Our method explicitly encodes type expectations during code generation and applies compiler feedback to resolve mismatches.

Other errors, although less frequent, such as invalid user types, redefinition, invalid variable, and unused label, were also addressed, which contributed to improving the overall compilation pass rate.

B. Threats to Validity

Internal Validity. While Rq2ST explicitly excludes requirements exhibiting literal substring matches with benchmark cases, we acknowledge the potential existence of semantically similar examples. This characteristic intentionally reflects the practical reality of case repositories in code generation systems [20, 34, 41], where semantic overlaps naturally occur.

Construct Validity. While standard quantitative metrics (Compilation Pass Rate, Average Error Count) [11, 34] provide initial validation, they are insufficient for guaranteeing functional correctness. To address the lack of publicly available test benchmarks, we implemented an industry-peer-reviewed automated test case generation framework with rigorous quality filtering. Our experimental results demonstrate the method's efficacy through high pass rates (over 76%).

External Validity. While our framework can theoretically support generating ST variants beyond Siemens' SCL and CODESYS' ST through vendor-specific configurations of Rq2ST and APILib, the generalization performance requires further investigation and empirical validation.

C. Limitations

Limited Semantic Checks. Our current approach primarily corrects code based on compilation feedback, which may miss runtime issues. Further improvements could incorporate time-sensitive, and safety-critical autonomous test suites or simulations—for instance, validating deterministic behavior, resource constraints, hardware configurations, and I/O mappings.

Limited Assessment of Functional Correctness, Safety, and Reliability. Our evaluation mainly focuses on metrics such as Compilation Pass Rate and Average Error Count (RQ1, RQ2), while for RQ3, experts assess correctness, conformance, modifiability, safety, and usefulness. However, the manual evaluation was exploratory and limited in scope, and we did not conduct a thorough assessment of functional correctness, safety, and reliability using more formal and rigorous methodologies.

Complex control logic and large-scale integration remain unexplored. The current benchmark tasks primarily focus on fundamental programming scenarios, while intricate control logics (e.g., interlocking mechanisms, recipe management) and large-scale system integration require further investigation.

VII. Related Work

Automating program generation from natural language (NL) requirements has long been a research objective. Early approaches to PLC program generation relied on formal methods, rule-based systems, and manually defined features [7, 47]. While effective in constrained domains, these methods required substantial human expertise and lacked adaptability to real-world industrial scenarios.

Recent advances in large language models (LLMs) [15, 39, 43] have shown promising code generation capabilities, with growing applications in industrial domains. Due to limited domain-specific training resources, most studies employ general-purpose LLMs directly. For instance, ChatGPT has been tested for IEC 61131-3 ST generation [26], while Koziolek and Koziolek [25] developed a P&ID-to-ST translation approach. However, these methods suffer from low determinism and poor compilation rates; our experiments show GPT-4o achieves only 35.47% success on ST generation tasks. Recently, a Siemens research group achieved 70% compilation success by first pretraining LLMs on OSCAT libraries and then refining them with generated high-quality samples, as validated on both OSCAT and the APPS Python benchmark.

To address these limitations, recent work has focused on improving generated code quality through human-AI collaboration and retrieval augmentation. LLM4PLC [11] introduced a user-guided iterative pipeline incorporating external verification, though its effectiveness varies with human expertise. Other approaches integrate RAG; Koziolek et al. [28] leveraged proprietary function blocks, while Agents4PLC [34] proposed a multi-agent framework combining LLMs with planning, RAG, and formal verification agents.

Key challenges remain in current methodologies: (1) the scarcity of high-quality, publicly available domain knowledge limits retrieval precision [28, 34]; (2) most systems target only specific ST variants, restricting generalizability; and (3) the absence of standardized benchmarks impedes objective comparison—existing evaluations range from case studies [25] to small-scale task sets (40 tasks in Fakih et al. [11], 23 in Liu et al. [34]). *AutoPLC addresses these by enabling vendor-aware generation, scaling evaluation to 945 tasks across two variants, and open-sourcing our domain knowledge base to advance the field.*

VIII. Conclusion

Industrial Structured Text development is often hampered by vendor dialect challenges. To address this issue, we introduced AutoPLC, an end-to-end framework for generating vendor-aware ST from NL requirement. AutoPLC integrates curated knowledge bases, task planning, retrieval-augmented generation, and industrial compiler-feedback improvement into a unified workflow. Our experiments on CODESYS' ST and Siemens' SCL datasets demonstrated that AutoPLC outperformed state-of-the-art baselines, particularly in compilation pass rates and error reduction. The importance of the core designs has been confirmed through ablation studies. Five ST experienced experts affirmed the usefulness of our generated code. In the future, we aim to tackle more complex logic and multi-device interfaces, and test on a broader set of real-world benchmarks derived from authentic engineering practices.

IX. Acknowledgement

REFERENCES

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[2] anthropic.com. Claude, 2024. URL https://claude.ai/new.

[3] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

[4] biendata.xyz. Generative AI Application Competition: Code Generation for Industrial Software - Biendata, June 2024. URL https://www.biendata.xyz/competition/siemens-ai.

[5] Tom B Brown. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[6] codesys.com. Codesys group, May 2021. URL https://www.codesys.com.

[7] Dániel Darvas, Enrique Blanco Viñuela, and István Majzik. Plc code generation based on a formal specification language. In *2016 IEEE 14th International Conference on Industrial Informatics*, pages 389–396. IEEE, 2016.

[8] Mario de Sousa. Source code repository, 2024. URL https://bitbucket.org/mjsousa/matiec_git. 2024-11-03.

[9] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Jingyuan Ma, et al. A survey on in-context learning, October 2024.

[10] Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. Self-collaboration code generation via chatgpt. *ACM Transactions on Software Engineering and Methodology*, 33(7):1–38, 2024.

[11] Mohamad Fakih, Rahul Dharmaji, et al. Llm4plc: Harnessing large language models for verifiable programming of plcs in industrial control systems. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*, pages 192–203, 2024.

[12] PLCopen for efficiency in automation. Iec 61131-3: Programming languages (edition 3.0-2013), 2024.

[13] Team GLM, Aohan Zeng, Bin Xu, Bowen Wang, et al. Chatglm: A family of large language models from glm-130b to glm-4 all tools. *arXiv preprint arXiv:2406.12793*, 2024.

[14] Qiuhan Gu. Llm-based code generation method for golang compiler testing. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, page 2201–2203. Association for Computing Machinery, 2023.

[15] Daya Guo, Qihao Zhu, Dejian Yang, et al. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 2024.

[16] Aaron Haag, Bertram Fuchs, Altay Kacan, and Oliver Lohse. Training llms for generating iec 61131-3 structured text with online feedback. *arXiv preprint arXiv:2410.22159*, 2024.

[17] Yue Huang, Lichao Sun, Haoran Wang, Siyuan Wu, Qihui Zhang, et al. Trustllm: Trustworthiness in large language models, 2024.

[18] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, et al. Qwen2.5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

[19] Mordor Intelligence. Plc market size & share analysis-growth trend & proecasts (2024-2029), 2024.

[20] Md Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. Mapcoder: Multi-agent code generation for competitive problem solving. *arXiv preprint arXiv:2405.11403*, 2024.

[21] Xue Jiang, Yihong Dong, Lecheng Wang, et al. Self-planning code generation with large language models. *ACM Transactions on Software Engineering and Methodology*, 33(7):1–30, 2024.

[22] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024.

[23] Fondazione Bruno Kessier. nuxmv, 2024. URL https://nuxmv.fbk.eu/.

[24] Georgiy Komarov. jubnzv/iec-checker, November 2024. URL https://github.com/jubnzv/iec-checker.

[25] Heiko Koziolek and Anne Koziolek. LLM-based Control Code Generation using Image Recognition. In *2024 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code)*, pages 38–45, April 2024.

[26] Heiko Koziolek, Sten Gruener, and Virendra Ashiwal. Chatgpt for plc/dcs control logic generation. In *2023 IEEE 28th International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8. IEEE, 2023.

[27] Heiko Koziolek, Virendra Ashiwal, Soumyadip Bandyopadhyay, and Chandrika K R. Automated control logic test case generation using large language models. In *2024 IEEE 29th International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8, 2024.

[28] Heiko Koziolek, Sten Grüner, Rhaban Hark, Virendra Ashiwal, et al. LLM-based and Retrieval-Augmented Control Code Generation. In *Proceedings of the 1st International Workshop on Large Language Models for Code*, LLM4Code '24, pages 22–29. Association for Computing Machinery, September 2024.

[29] Huayang Li, Yixuan Su, Deng Cai, Yan Wang, and Lemao Liu. A survey on retrieval-augmented text generation. *arXiv preprint arXiv:2202.01110*, 2022.

[30] Jia Li, Ge Li, Yongmin Li, and Zhi Jin. Structured chain-of-thought prompting for code generation. *ACM Transactions on Software Engineering and Methodology*, 2023.

[31] Xiaoli Lian, Shuaisong Wang, Jieping Ma, Xin Tan, Fang Liu, et al. Imperfect code generation: Uncovering weaknesses in automatic code generation by large language models. In *2024 IEEE/ACM 46th International Confer-*

ence on Software Engineering: Companion Proceedings (ICSE-Companion), pages 422–423, 2024.

[32] Xiaoli Lian, Shuaisong Wang, Hanyu Zou, Fang Liu, Jiajun Wu, and Li Zhang. Incorporating verification standards for security requirements generation from functional specifications. *Proc. ACM Softw. Eng.*, 2(FSE), June 2025.

[33] Dianshu Liao, Shidong Pan, Qing Huang, Xiaoxue Ren, Zhenchang Xing, Huan Jin, and Qinying Li. Context-aware code generation framework for code repositories: Local, global, and third-party library awareness. *arXiv preprint arXiv:2312.05772*, 2023.

[34] Zihan Liu, Ruinan Zeng, et al. Agents4plc: Automating closed-loop plc code generation and verification in industrial control systems using llm-based agents. *arXiv preprint arXiv:2410.14209*, 2024.

[35] Meta. Llama-3.1 (8b): A large language model, 2024.

[36] Fan Mo, Miriam Ugarte Querejeta, Joseph Hellewell, Hamood Ur Rehman, Miren Illarramendi Rezabal, Jack C. Chaplin, David Sanderson, and Svetan Ratchev. Plc orchestration automation to enhance human–machine integration in adaptive manufacturing systems. *Journal of Manufacturing Systems*, 71:172–187, 2023. ISSN 0278-6125. doi: https://doi.org/10.1016/j.jmsy.2023.07. 015. URL https://www.sciencedirect.com/science/article/ pii/S0278612523001474.

[37] M. Niang, B. Riera, A. Philippot, J. Zaytoon, F. Gellot, and R. Coupat. A methodology for automatic generation, formal verification and implementation of safe plc programs for power supply equipment of the electric lines of railway control systems. *Computers in Industry*, 123:103328, 2020. ISSN 0166-3615. doi: https://doi.org/10.1016/j.compind.2020. 103328. URL https://www.sciencedirect.com/science/ article/pii/S0166361520305625.

[38] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, et al. Codegen: An open large language model for code with multi-turn program synthesis. In *ICLR*. OpenReview.net, 2023.

[39] OpenAI. Chatgpt: Optimizing language models for dialogue, 2022.

[40] oscat.de. Oscat basic. product description and specifications, 2024-11-03. URL http://www.oscat.de/de/ 63-oscat-basic-321.html.

[41] Md Rizwan Parvez, Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Retrieval augmented code generation and summarization. *arXiv preprint arXiv:2108.11601*, 2021.

[42] Stephen Robertson and Hugo Zaragoza. The probabilistic relevance framework: Bm25 and beyond. *Found. Trends Inf. Retr.*, 3(4):333–389, April 2009.

[43] Baptiste Roziere, Jonas Gehring, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

[44] Jon Saad-Falcon, Omar Khattab, Keshav Santhanam, et al. UDAPDR: Unsupervised domain adaptation via LLM prompting and distillation of rerankers. In *Pro-*

ceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, pages 11265–11279. Association for Computational Linguistics, December 2023.

[45] Siemens. Library of general functions (LGF) for SIMATIC STEP 7 (TIA Portal) and SIMATIC S7... - ID: 109479728 - Industry Support Siemens, 2024.

[46] Siemens. SIMATIC STEP 7 inkl. Safety, S7-PLCSIM and WinCC V19 TRIAL Download - ID: 109820994 - Industry Support Siemens, 2024.

[47] Michael Steinegger and Alois Zoitl. Automated code generation for programmable logic controllers based on knowledge acquisition from engineering artifacts: Concept and case study. In *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation*, pages 1–8. IEEE, 2012.

[48] technavio. Programmable logic controller (plc) market analysis apac, north america, europe, middle east and africa, south america - us, china, japan, germany, uk - size and forecast 2024-2028, 2024.

[49] Yuchen Tian, Weixiang Yan, Qian Yang, Qian Chen, Wen Wang, et al. CodeHalu: Code Hallucinations in LLMs Driven by Execution-based Verification, April 2024.

[50] WEVOLVER. Plc programming: A comprehensive guide to mastering the art of automation, 2024.

[51] Tongtong Wu, Weigang Wu, Xingyu Wang, Kang Xu, Suyu Ma, Gholamreza Haffari, et al. Versicode: Towards version-controllable code generation, 2024.

[52] Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. Learning to mine aligned code and natural language pairs from stack overflow. In *International Conference on Mining Software Repositories*, MSR, pages 476–486. ACM, 2018.

[53] Shun Zhang, Zhenfang Chen, Yikang Shen, Mingyu Ding, Joshua B. Tenenbaum, and Chuang Gan. Planning with large language models for code generation. In *The Eleventh International Conference on Learning Representations*, 2023.

[54] Xiangyu Zhang, Yu Zhou, Guang Yang, and Taolue Chen. Syntax-aware retrieval augmented code generation. In *The 2023 Conference on Empirical Methods in Natural Language Processing*, 2023.

[55] Penghao Zhao, Hailin Zhang, et al. Retrieval-Augmented Generation for AI-Generated Content: A Survey, February 2024.