Automated PLC Programming Using a Multi-Task LLM System

Teresa Cacciapaglia*
Department of Electrical and Information Engineering
Polytechnic University of Bari, 70126
Bari, Italy
t.cacciapaglia@phd.poliba.it
*Corresponding author

Denis Ruffino
AROL S.p.A., 14053
Canelli, Italy
denis.ruffino@arol.com

David Naso
Department of Electrical and Information Engineering
Polytechnic University of Bari, 70126
Bari, Italy
david.naso@poliba.it

Paolo Roberto Massenio
Department of Electrical and Information Engineering
Polytechnic University of Bari, 70126
Bari, Italy
paoloroberto.massenio@poliba.it

***Abstract*—Programmable Logic Controllers (PLCs) are central to industrial automation, yet their programming remains largely manual and time-consuming. While Large Language Models (LLMs) excel at general-purpose code generation, their use in industrial control is limited by tight coupling with physical hardware, constrained I/O signals, and strict safety requirements. We propose a novel approach to PLC code generation that takes as input structured lists of sensors and actuators from electrical CAD tools, together with the required control functionality. The task is formulated as generating machine-specific control functions grounded in available signals, ensuring physical implementability. The system is based on a single LLM adapted through multi-stage fine-tuning and enhanced with Retrieval-Augmented Generation (RAG). Experiments on error detection, code fixing, and code generation show that fine-tuning significantly improves performance. RAG provides strong gains in error detection and code generation, while offering limited benefit for code fixing, highlighting its task-dependent effectiveness.**

***Index Terms*—PLC code generation; IEC 61131-3; Structured Text (SCL); LLM; Industrial automation**

I. Introduction

Programmable Logic Controllers (PLCs) underpin safety-critical industrial machines and production lines, forming the backbone of industrial automation. However, programming PLCs under the IEC 61131-3 standard [1] (e.g., Structured Text, Ladder Logic) remains largely manual, time-consuming, and dependent on expert knowledge. While Large Language Models (LLMs) have achieved strong performance in general-purpose code generation (e.g., Python, C++) [2], [3], their application to industrial control systems remains limited.

Recent works such as Agents4PLC [4], LLM4PLC [5], and AutoPLC [6] have begun exploring this direction. Agents4PLC demonstrates the effectiveness of decomposing the task into planning, generation, and verification stages, while other approaches focus on fine-tuning models for PLC-specific languages. On the other hand, PLC programming introduces unique challenges: code is tightly coupled to physical hardware, constrained by the available I/O signals, and subject to strict safety requirements. These constraints require domain-specific adaptations beyond standard code generation techniques.

In industrial practice, a recurring inefficiency arises when new machines share functional similarities with existing ones. Although control logic often follows similar patterns, due to the reuse of sensors and actuators in comparable configurations, big portions of PLC code are typically rewritten from scratch for each variant. This leads to unnecessary engineering effort, especially in production environments where families of related machines are deployed.

In this work, we address this problem by targeting the automatic generation of SCL (Structured Control Language) code for Siemens TIA Portal [7]. Instead of relying on natural language specifications, we use structured component lists exported from electrical CAD tools (e.g., SPAC Automation [8]), which provide a compact representation of the sensors and actuators available in a machine. This formulation shifts the problem from generic code generation to the synthesis of machine-specific control functions (e.g., alarms, limit switches, operating modes), constrained by the available I/O configuration and the required functionalities. As a result, the generated code is not only syntactically valid, but also grounded in the physical system and directly implementable. To tackle this problem, we use a unified approach based on a single LLM progressively adapted through multi-stage fine-tuning and enhanced with retrieval capabilities.

The main contributions of this paper are as follows:

- We propose a novel framework for PLC code generation that takes as input structured lists of available sensors and actuators, exported from electrical CAD tools, together with the required control functionality. The task is formulated as the generation of specific control functions

grounded in physically available signals, rather than as generic PLC code synthesis from natural language.
- We design a unified multi-stage adaptation pipeline based on a single LLM. The model is first specialized for signal understanding, so as to associate PLC tags with their semantic role and software metadata, and is then further trained in a multi-task setting on code generation, error detection, code correction, and code completion.
- We integrate a Retrieval-Augmented Generation (RAG) mechanism [9] that enriches inference with two sources of knowledge: machine-specific signal databases and reference SCL control functions. This improves the consistency of generated outputs with the available I/O configuration and with recurring programming patterns.
- We provide an experimental evaluation on three PLC-oriented tasks (error detection, code fixing, and code generation) comparing a base model, a Low-Rank Adaptation (LoRA) [10] fine-tuned model, and a LoRA+RAG configuration. The results show that fine-tuning substantially improves structural correctness and performance, while the benefit of retrieval is task-dependent: RAG provides strong gains in tasks such as error detection and code generation, where access to external signal and function knowledge is beneficial, but does not improve code fixing, where retrieval may introduce distracting patterns.

The remainder of this paper is organized as follows. Section II presents the proposed framework, Section III describes the dataset and training setup, Section IV provides the experimental evaluation, and Section V concludes the paper.

II. Proposed Approach

The proposed system is built around a single LLM acting as a unified agent. The model is progressively adapted to PLC programming through a multi-stage fine-tuning and enhanced with retrieval capabilities. The goal is to enable a single model to handle the full spectrum of PLC-related tasks, including signal interpretation, code generation, and error detection and correction. The approach is structured in three stages: (i) signal understanding, where the model is fine-tuned to interpret and reason over PLC signals, learning to associate each tag (sensor, actuator, alarm, etc.) with its functional semantics and metadata; (ii) multi-task code generation and correction, where the model is jointly trained on error detection, code correction, code completion, and code generation from natural language requirements; (iii) integration of RAG, where the fine-tuned model access a database of signals and reference SCL functions during inference. Throughout this section, an automatic capping machine is used as a representative industrial scenario.

A. Signal Understanding

In the first stage, a dedicated dataset is constructed by combining two sources: Siemens TIA Portal variable tables (XML) and SPAC Automation component lists (CSV) from an electrical schematic of an automatic machine. The CSV provides component identifiers and functional roles, while the XML supplies metadata such as data type, address, and memory area. Merging them links each physical component to its functional meaning and software representation, which is essential for generating correct SCL code.

This dataset is fundamental for generalization: each sensor is identified by a unique tag belonging to a predefined set, allowing the model to recognize familiar signal patterns and, when presented with unseen sensor, it should be able to embody it in the correct context. This capability is critical in practice, as new machine variants often introduce a small number of novel components alongside a largely known set. For example, a new capping machine variant may add sensors for punch height regulation to an otherwise standard configuration. The model must be able to handle both known and novel signals, generating the appropriate control logic for each.

The dataset is manipulated and converted into Alpaca-style [11] instruction tuning format. For each signal, three categories of instruction/input/output pairs are included:

1) Metadata Extraction: The model is trained to map a PLC tag to its structured metadata representation. Here's an example with the sensor that we will call Height1, which measures the turret height of a capping machine:

"instruction": "Given a Siemens TIA Portal PLC tag, return its metadata as JSON (JSON only).",
"input":"TAG: Height1",
"output":"io_kind": "INTERNAL",
"data_type": "Bool",
"logical_address": "M60.1",
"area": "M"

The main objective of tuning the LLM with this dataset stands in its future capability: when receiving for instance the tag Height2, the model will categorize it in a similar way.

2) Signal Selection from Context: The model receives a list of available signals together with a functional requirement expressed in natural language. The objective is to select the most appropriate PLC tag that satisfies the requirement. Here's an example where we want to select an alarm sensor:

"instruction": "Given a list of available signals and a requirement, reply ONLY with the most suitable PLC tag from the list.",
"input":"AVAILABLE SIGNALS:
- Height1 (INTERNAL Bool %M84.7)
- Check_bottle1 (INTERNAL Bool %M502.3)
- Alarm1 (INTERNAL Bool %M100.1)
- Stop_emergency (INTERNAL Bool %M121.7)
- ...
REQUIREMENT: ALARM."
"output":"Alarm1"

This task enables the model, when writing code is required, to chose between a list of available signals that come from the machine's electrical schematics (e.g., Height1, Check_bottle1, Alarm1). Given a requirement such as "ALARM," the model is expected to identify the correct tag (e.g., Alarm1).

3) Tag Classification among Candidates: The model has to choose the correct tag based on the requirement, without knowing its characteristics. For example, here the model has to chose the sensor CL_1 from the list, as we asked him to select a Clock of 1 second:

"instruction": "Choose the correct PLC tag among the options. Reply ONLY with the tag.",
"input":"REQUIREMENT: Clock 1 s
OPTIONS: Height1, KC9, AL_1, CL_1"
"output":"CL_1"

B. Multi-Task Code Generation and Correction

To generate correct control logic, a second fine-tuning stage is introduced, in which the same model is trained on a multi-task dataset. This dataset is constructed from a set of representative control functions covering common programming patterns. These functions act as a domain-specific knowledge base, allowing the model to learn how to produce syntactically correct and semantically consistent SCL code. For each base function, multiple instruction–input–output examples are generated and organized into four task categories:

1) Error Detection: The model is required to identify if a piece of PLC code contains errors (syntactic or logical), e.g.,

"instruction": "Detect the syntax error in the following Siemens SCL/ST code. Reply in this exact format: ERROR: <short name> (DETAIL: <one sentence>)",
"input":"//MOD_AUT function:
VAR
    M_AUT   :   BOOL;
    G23   :   BOOL;
END_VAR
IF M_AUT THEN
    G23 = TRUE;
ELSE
    G23 := FALSE;
END_IF"
"output":"ERROR: Wrong assignment operator (= instead of :=). "

2) Code Fixing: The model receives faulty PLC code and must produce a corrected version that is syntactically valid and consistent with the intended functionality, e.g.,

"instruction": "Fix the syntax errors in the following Siemens SCL/ST code. Preserve behavior. Output corrected code only.",
"input":"//MOD_AUT function:
VAR
    M_AUT   :   BOOL;
    G23   :   BOOL;
END_VAR
IF M_AUT THEN
    G23 = TRUE;
ELSE
    G23 := FALSE;"
"output":"
VAR
    M_AUT   :   BOOL;
    G23   :   BOOL;
END_VAR
IF M_AUT THEN
    G23 := TRUE;
ELSE
    G23 := FALSE;
END_IF"

3) Code Completion: The model is fed with a part of a function and has to complete the code, e.g.,

"instruction": "Complete the Siemens SCL/ST code. Output ONLY the missing continuation (do not repeat the prefix).",
"input":"//MOD_AUT function:

Prefix:
VAR
    M_AUT   :   BOOL;
    G23   :   BOOL;
END_VAR

Continue the code from here:
"output":"
IF M_AUT THEN
    G23 := TRUE;
ELSE
    G23 := FALSE;
END_IF"

4) Code Generation: The model receives a functional requirement and a list of available signals, and must generate PLC code using only those signals and following Siemens SCL syntax. For instance:

"instruction": "Write Siemens TIA Portal SCL/ST code that implements the requested function using ONLY the available signals. Output code only.",
"input":"Requested function:
automatic mode activation.
Available signals: T23 (Internal Bool), EN2 (Internal Bool), CL1 (Internal Clock), G23 (Internal Bool), K34 (Internal Bool), M_AUT (Internal Bool), WR4 (Internal Bool). "
"output":"//MOD_AUT function:
VAR
    M_AUT   :   BOOL;
    G23   :   BOOL;
END_VAR
IF M_AUT THEN
    G23 := TRUE;
ELSE
    G23 := FALSE;
END_IF"

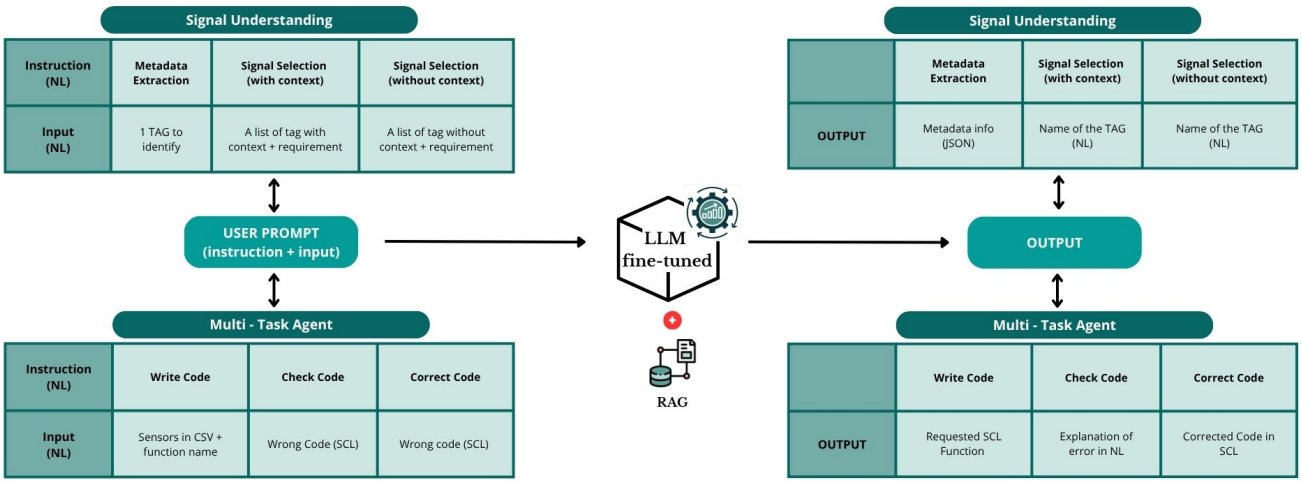

Fig. 1. Structure of the proposed framework.

C. Retrieval Augmented Generation

The third step augments the fine-tuned model through a RAG mechanism [9]. The Multi-Task agent is enhanced with a Retrieval Agent, that dynamically queries two dedicated knowledge bases at inference time. The first database aggregates the signal-level information described in Section II-A, comprising the full catalog of sensors and actuators available across a given machine family. The second database collects a set of reference control functions already implemented in SCL language, covering the most representative programming patterns of the target machine family, for instance in case of capping machines: functions to regulate the turret, the punch and the piston height, activate the manual mode, etc.

D. Framework Overview and Design Considerations

Figure 1 summarizes the overall structure of the proposed framework. The system is designed to generate individual control functions rather than complete PLC programs. The model can support automation engineers in recurring and well-defined tasks, reducing manual coding effort while maintaining reliability. A key aspect of the proposed approach is the use of a single multi-task model instead of multiple specialized agents. Multi-task training enables different capabilities while improving overall code quality, as learning error detection and correction helps the model internalize common failure patterns and generate more robust code. Unlike generic LLM-based code generation approaches, the proposed framework operates on structured inputs extracted from industrial CAD tools.

At inference time, the same model can be prompted for code generation, error detection, and correction, with these roles emerging from a unified representation rather than separate agents [4]. In the current implementation, these capabilities are evaluated sequentially, making the framework a first step toward iterative generate–verify–correct closed-loop pipelines.

TABLE I
DATASET COMPOSITION AND TRAIN/VALIDATION SPLIT

Stage / Task	Total	Train	Validation
Signal Understanding			
Metadata Extraction	360	340	20
Signal Selection	360	340	20
Tag Classification	360	340	20
Multi-Task Training			
Error Detection	60	50	10
Code Fixing	60	50	10
Code Completion	36	30	6
Code Generation	60	50	10
Total	1296	1200	96

III. Experimental Setup

This section outlines the employed dataset alongside the implemented fine-tuning and RAG mechanism setups.

A. Employed Dataset

The datasets used for fine-tuning are derived from representative industrial control functions and associated signal descriptions provided by an industrial partner (Arol S.p.A.).

For the signal understanding stage, the dataset includes three sample types: (i) metadata extraction from PLC tags (360), (ii) signal selection given a set of available signals and a functional requirement (360), and (iii) tag classification based on requirements (360). For the multi-task stage, the dataset is built from 12 base control functions. Each function generates multiple examples across the four tasks: 5 for error detection, 5 for code fixing, 3 for code completion, and 5 for code generation, for a total of 216 samples. Table I summarizes the dataset composition and the train/validation split.

B. Fine-Tuning Setup

The backbone of the proposed system is Qwen2.5-Coder 3B-Instruct model, selected for its strong performance in

code generation tasks and its compatibility with efficient fine-tuning techniques [12]. The model is retrieved from Hugging Face [13] and fine-tuned using the LLaMA-Factory framework [14], which provides a modular environment for instruction tuning across different LLM architectures. To reduce computational cost while maintaining performance, we adopt Low-Rank Adaptation (LoRA) [10]. Instead of updating all model parameters, LoRA introduces a small set of trainable low-rank matrices into selected layers while keeping the others frozen. This significantly reduces the number of trainable parameters, enabling domain adaptation on limited hardware.

Fine-tuning is performed for three epochs using the AdamW optimizer [15], with a learning rate of 1×10^{-4}, beta coefficients (0.9, 0.999), and $\epsilon = 10^{-8}$, combined with a cosine scheduler and 3% warmup. Training converges rapidly and stabilizes after the initial steps, as shown in Figure 2, which reports the evolution of the training loss over the full dataset of Table I.

C. RAG Implementation

The RAG mechanism operates over two complementary knowledge sources. The first is a tag database derived from a representative machine which comprises approximately 200 PLC variables, representing physical signals typically found in the relative electrical schematic. The retrieving action on the signal understanding is useful to provide the code correction/generation stage with machine-specific variable names associated with their functionalities. The second source collects a set of ten SCL functions covering the most representative programming patterns of the target machine family, different from the ones used for the fine-tuning.

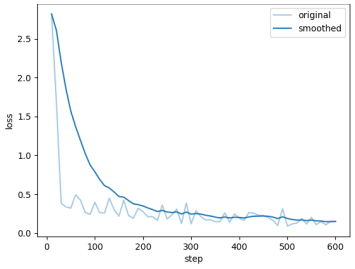

Fig. 2. Loss with the fine-tuning of Qwen2.5-Coder-3B-I over the dataset.

IV. Performance Evaluation

This section compares three system configurations across three tasks (error detection, code fixing, and code generation): BASE (pre-trained Qwen2.5-Coder-3B), LoRA (fine-tuned without retrieval), and LoRA+RAG (fine-tuned with retrieval).

A. Error Detection

Table II reports the results for the error detection task in terms of accuracy, macro-F1, and output format compliance. Accuracy measures the percentage of correctly predicted error labels, while macro-F1 averages the F1 score across all error classes to account for class imbalance. Output format compliance evaluates whether the model produces a structurally valid response matching the expected schema, i.e., an output composed of two lines starting with `ERROR:` and `DETAIL:`. A clear performance progression can be observed across the three evaluated settings.

TABLE II
Error Detection Results

Setting	acc	F1	format
BASE	19.5	11.5	35.3
LoRA	28.1	20.3	51.3
LoRA+RAG	**73.4**	**70.9**	**100**

B. Code Fixing

For the code fixing task, three difficulty levels have been defined to better assess the performances: *easy* (simple functions with only one easily identifiable error, e.g., '=' instead of ':=', missing END_IF), *medium* (more structured control logic with more difficult errors such as incorrect logic operators), and *hard* (complex functions with many errors and interacting mistakes that require reconstructing the intended control logic). Tables III show that LoRA significantly improves code fixing performance across all difficulty levels, largely outperforming the base model in both F1 and similarity while achieving perfect format compliance. Here, the second metric (sim) measures the character-level similarity between prediction and ground truth. The format compliance is evaluated by checking whether the generated code contains core ST constructs, such as control structures (e.g., IF, END_IF) and/or variable declarations (VAR blocks), ensuring that the output resembles valid executable code. RAG does not improve performance in this task and slightly degrades results compared to LoRA alone, suggesting that retrieval may introduce irrelevant or conflicting patterns during correction. However, format compliance remains at 100%, indicating that RAG affects semantic accuracy rather than structural correctness of model's output.

TABLE III
Code Fixing Results

	F1	**sim**	**fmt**
Easy			
BASE	30.4	32.4	81.0
LoRA	**76.1**	**78.7**	**100**
LoRA+RAG	73.4	70.9	**100**
Medium			
BASE	32.0	42.6	79.3
LoRA	**76.5**	**80.0**	**100**
LoRA+RAG	69.2	62.6	**100**
Hard			
BASE	28.9	31.6	57.1
LoRA	**54.3**	**58.3**	**100**
LoRA+RAG	44.8	42.2	**100**

C. Code Generation

In the code generation task, two difficulty levels are considered: *easy*, involving simple functions with basic boolean

logic (e.g., switching a light on/off), and *medium*, involving more structured logic with multiple conditions and signal coordination (e.g., piston height control).

When considering the code generation, Table IV highlights a different trend compared to the fixing task. While LoRA alone doesn't improve performance, degrading F1 scores, adding RAG leads to substantial improvements across all metrics. Here, VAR represents the percentage of cases in which the variable block is correctly present and properly initialized (i.e., correctly defined within the VAR ... END_VAR section). In the code generation task, format compliance is defined more strictly. It requires the presence of a correctly structured program, including a valid VAR ... END_VAR block and/or a program/function declaration (e.g., PROGRAM, FUNCTION), as well as consistent use of ST syntax (e.g., logical operators such as AND, OR, and assignments with :=). Code generation strongly benefits from retrieval and performances could be improved with a larger dataset of functions.

TABLE IV
CODE GENERATION RESULTS

Setting	**Easy**			**Medium**		
	F1	VAR	fmt	F1	VAR	fmt
BASE	39.4	24.5	45.3	21.4	20.8	39.9
LoRA	24.1	25.6	56.2	14.4	18.9	30.2
LoRA+RAG	**44.5**	**85.7**	**100**	**63.9**	**42.9**	**100**

V. CONCLUSIONS

This work presented a novel approach to PLC code generation grounded in structured signal data rather than the introduction of new LLM techniques. Instead of relying on natural language specifications, the proposed framework generates machine-specific control functions from lists of available sensors and actuators. A single LLM is progressively fine-tuned through multiple stages and enhanced with retrieval mechanism, enabling signal understanding, code generation, and error detection/correction within a unified model. Experimental results show that fine-tuning significantly improves performance across all tasks, while retrieval enhances error detection and code generation, improving both accuracy and format compliance, although it provides limited benefits for code fixing.

Overall, the results represent promising preliminary evidence for the proposed approach. Despite being trained on a relatively small, domain-specific dataset focused on individual control functions, it already demonstrates the potential of combining fine-tuning and retrieval for PLC programming tasks. The dataset, although limited, is structured, curated, and representative of real industrial patterns. Current evaluation metrics offer useful insights but are not sufficient for safety-critical validation, which would require formal verification and compilation checks. No formal statistical analysis was performed, as the observed improvements were large and consistent across tasks.

Building on these, future work will extend the framework toward fully automated PLC programming, including complete machine logic generation. A key direction is the integration of a closed-loop generate–verify–correct pipeline. Additional efforts will focus on larger and more diverse datasets, improved robustness to unseen configurations, cross-vendor support, and the integration of formal verification. Moreover, we acknowledge the importance of comparing our work with existing frameworks (e.g. AutoPLC, Agents4PLC):ongoing work includes benchmarking against these recent PLC-specific LLM systems.

REFERENCES

[1] International Electrotechnical Commission, "IEC 61131-3:2013 Programmable Controllers - Part 3: Programming Languages," IEC," International Standard, 2013. [Online]. Available: https://webstore.iec.ch/en/publication/4552

[2] K. Zhang, J. Li, G. Li, X. Shi, and Z. Jin, "Codeagent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, 2024, p. 13643–13658.

[3] Z. Chen, S. Fang, and M. Monperrus, "Supersonic: Learning to generate source code optimizations in C/C++," *IEEE Transactions on Software Engineering*, 2024.

[4] Z. Dang, Y. Wang, Z. Liu, F. Dong, J. Chen, J. Tan, Y. Liu, J. Zhu, and L. Wang, "Agents4plc: Shielded LLM agent for sample-efficient industrial control program generation," *arXiv preprint arXiv:2410.14209*, 2024. [Online]. Available: https://arxiv.org/abs/2410.14209

[5] M. Fakih, R. Szsz, Y. Cheng, M. Siegel, S. Buechner, A. Fay, M. Wollschlaeger, and B. Vogel-Heuser, "LLM4PLC: Harnessing large language models for verifiable programming of PLCs in industrial control systems," in *Proceedings of the IEEE International Conference on Automation Science and Engineering (CASE)*. IEEE, 2024, pp. 1–8.

[6] W. Zhang, J. Li, M. Chen, and X. Wang, "AutoPLC: Automated IEC 61131-3 structured text code generation for PLCs using large language models," *IEEE Transactions on Industrial Informatics*, vol. 20, no. 5, pp. 7123–7134, 2024.

[7] Siemens AG, *TIA Portal Openness: Automating Engineering Workflows*, Siemens Digital Industries, 2024. [Online]. Available: https://support.industry.siemens.com/cs/document/109792244/

[8] SDProget, "Spac automazione," Online, accessed: 2026-02-16. [Online]. Available: https://sdproget.it/prodotti/automazione/spac-automazione/

[9] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, *et al.*, "Retrieval-augmented generation for knowledge-intensive NLP tasks," in *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*, 2020, pp. 9459–9474. [Online]. Available: https://arxiv.org/abs/2005.11401

[10] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "LoRA: Low-rank adaptation of large language models," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2022. [Online]. Available: https://arxiv.org/abs/2106.09685

[11] R. Taori, I. Gulrajani, T. Zhang, Y. Dubois, X. Li, C. Guestrin, P. Liang, and T. B. Hashimoto, "Stanford Alpaca: An instruction-following LLaMA model," https://github.com/tatsu-lab/stanford_alpaca, 2023.

[12] B. Hui, J. Yang, Z. Cui, and J. Yang, "Qwen2. 5-coder technical report."

[13] Hugging Face, "HuggingFace: The ai community building the future," https://huggingface.co, 2024, online platform for machine learning models, datasets, and applications.

[14] Y. Zheng, R. Zhang, J. Zhang, Y. Ye, Z. Luo, and Y. Ma, "LLaMA-Factory: Unified efficient fine-tuning of 100+ LLMs," https://github.com/hiyouga/LLaMA-Factory, 2024, gitHub repository.

[15] PyTorch Contributors, "torch.optim.adamw," https://docs.pytorch.org/docs/stable/generated/torch.optim.AdamW.html, 2024, accessed: 2026-04-24.