ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DOTTORATO DI RICERCA IN

COMPUTER SCIENCE AND ENGINEERING

Ciclo 38

Settore Concorsuale: 01/B1 - INFORMATICA

Settore Scientifico Disciplinare: INF/01 - INFORMATICA

CREATING A CODE ASSISTANCE TOOL USING LOW COMPUTATIONAL RESOURCES

Presentata da: Adnan Riaz

Coordinatore Dottorato

Paola Salomoni

Supervisore

Moreno Marzolla

Co-supervisore

Davide Rossi

Esame finale anno 2026

Acknowledgements

Finanziato dall'Unione europea – Next Generation EU, Missione 4, Componente 2, Investimento 3.3 (D.M. 352/2022) CUP J33C22001400009

Contents

List of Figures

List of Tables

Abstract

Code Large Language Models have the ability to automate programming tasks and help developers work faster, but applying them in industrial settings is not straightforward due to limited resources, strict security rules, and a lack of domain-specific data, which make standard approaches impractical. This doctoral research investigates the development of a coding assistant using limited computational resources, conducted in response to a problem stated by *Coesia*, a leading industrial firm of automation and packaging solutions. The research addresses the need for an in-house, privacy-preserving, and resource-efficient code assistant, capable of supporting the Structured Text programming language defined by the IEC 61131-3 standard. Given the limited availability of domain-specific data and the stringent confidentiality policies typical of industrial environments, this study explores both the technical and organizational dimensions of how to build lightweight, specialized LLMs that perform effectively in industrial settings. It first presents a feasibility study of *PEFTpyCODER*, a Parameter-Efficient Fine-Tuned model developed as an in-house code assistant using limited computational resources, therefore eliminating the need of Cloud resources. This study employs Parameter-Efficient Fine-Tuning (PEFT), Low-Rank Adaptation (LoRA), and 4-bit quantization techniques for fine-tuning an open source (Phi-3-mini-128k-instruct) model. The proposed model demonstrated superior performance on two widely accepted Python benchmarks (MBPP and HumanEval) compared to state-of-the-art code LLMs that are up to 5 times larger. Building on this, a synthetic Structured Text (ST) code generation pipeline is designed to create a high-quality, domain-specific dataset for Programmable Logic Controller (PLC) programming by integrating Azure OpenAI GPT-4 instance, automatic compilation and validation using TwinCAT, and semantic verification through the TcUnit (mutation testing) framework. This pipeline enables the large-scale creation and validation of a syntactically and functionally correct ST code dataset, forming the foundation for model training and evaluation. This work also examines the licensing and innovation management aspects of deploying LLMs in industrial settings, providing a broader organizational perspective. Finally, this work reports experimental results on open-source code LLMs for the ST code generation tasks using few-shot prompting, both without and with prompt optimization (Prompt Engineering), and also fine-tuning of open-source code LLMs using PEFT, LoRA, and quantization techniques for ST code completion tasks.

The initial results obtained from the experiments demonstrate the technical feasibility and performance of the proposed approach.

Keywords: Code Generation; Limited Computational Resources; Structured Text (ST); Code Large Language Models (Code LLMs); Parameter-Efficient Fine-Tuning (PEFT); Low-Rank Adaptation (LoRA); In-House Code Assistant; Licensing and Innovation Management

Chapter 1

Introduction

1.1 Background of the Study

Industrial automation has reshaped modern manufacturing by allowing precise, efficient, and scalable control over machines and processes. In most industrial systems, Programmable Logic Controllers (PLCs) are the heart of industrial automation systems and are specialized computing devices which execute control programs with high reliability and real-time responsiveness. Because of its strength, deterministic nature, and capability to communicate with industrial data communication systems, PLCs have become popular in many industries [18] including manufacturing and packaging solutions, where I have observed the vital role of PLCs in ensuring smooth production operations such as companies like Coesia S.p.A.

A key component of PLC-based control systems is Structured Text (ST), which is one of the five standard PLC programming languages defined by IEC 61131-3. ST is a Pascal-like language, high level, meant to write in a readable and structured way complex logic and arithmetic operations[30]. Because of its expressive syntax and the proximity to general-purpose programming languages, it is especially well-suited for expressing the complex logic of industrial automation. In my experience during field visits and literature review, ST is particularly valuable for describing the sophisticated logic found in packaging machinery, where multiple sensors, actuators, and production sequences must be coordinated accurately. The need for scale development, validation, and deployment of ST-based automation logic has become much more critical as industrial systems become more interconnected and move towards Industry 4.0.

At the same time, recent development of LLMs like GPT-4[4], PaLM[25], and LLaMA[108] have brought a paradigm shift in the field of Artificial

Intelligence. These models are trained on large sets of natural language as well as programming languages and have proven exceptional abilities in tasks such as code generation, summarization, and error correction[23]. With the success of LLMs in general purpose code generation, interest of using LLM to generate domain-specific code (such as automation scripts and PLC code) has been prompted.

However, in comparison with other programming languages like Python, Java, C++, or Java Script, where a large number of high-quality datasets and benchmarks (e.g., CodeXGLUE[72], HumanEval[23], MBPP[11]) are present, allowing the fine-tuning and evaluation of LLMs. The field of industrial automation specifically Structured Text programming, which is hampered by the absence of high-quality publicly available datasets. Such shortage poses a major challenge to the utilization of LLMs in industrial code generation and analysis. Industrial projects are usually proprietary, and there are also intellectual property limitations and syntax-specific domains making the gathering and unrestricted sharing of ST code even more difficult. Furthermore, there are lack of standard evaluation tools for ST code generated by AI models. Ensuring that generated code is syntactically correct, functionally valid, and industrially safe remains a significant challenge as well. Without proper datasets and evaluation pipelines, progress in AI-driven industrial automation is slow and fragmented.

Recognizing these gaps, this thesis focuses on developing a framework for leveraging LLMs to generate, validate, and evaluate ST code. The goal is to produce both methodological and practical contributions: creating resources that can support further research and providing industrially relevant tools for companies like Coesia, where efficient and reliable ST code generation is critical for reducing engineering effort and accelerating production line deployment.

1.1.1 Industrial Automation and the Role of PLCs

Industry 4.0 or the fourth industrial revolution has introduced a new era of innovation in the manufacturing and production processes[19]. This paradigm shift includes cyber-physical systems, Internet of Things (IoT), and other data analytics in the traditional industrial setting[63][77]. One of the most important facilitators of this change is the industrial automation that strives to minimize human influence, increase productivity, and guarantee system reliability and safety. Most industrial automation systems have PLCs embedded, real-time computing devices used to automate electromechanical processes.

The first introduction of PLCs was in the early 70s to replace the hard-

wired relay-based systems. They have since then become intelligent, modular, and programmable platform which can perform a wide range of automation jobs like sequencing, timing, counting, logic control, and data processing. PLCs are designed to be used in hostile industrial environments, vibration, humidity, electromagnetic interferences, and offer determinism to the machine which is critical to ensuring safety and real-time operation[18].

In Coesia, PLCs are part of packaging lines, assembly stations, and process monitoring systems. For example, in case of a typical packaging machine, the PLC coordinates conveyor belts, robotic arms, sensors, and actuators in precise sequences. Such systems only work with a successful hardware and logic programmed into the PLC. Any errors or inefficiencies in this logic can lead to downtime, reduced throughput, or even safety incidents.

The ability to use standardized programming languages, as defined by the IEC 61131-3 standard, is one of the main characteristics that has seen PLCs become an all-powerful technology in terms of industrial automation and manufacturing. This standard defines five programming paradigms of PLCs: Ladder Diagram (LD), Function Block Diagram (FBD), Instruction List (IL), Sequential Function Chart (SFC), and Structured Text (ST). ST has become popular because of its expressive capabilities and resemblance with high-level programming languages. PLC programming is still largely manual and labor-intensive, despite of its importance. Skilled engineers are required to debug, write, and maintain complex control programs. This reliance on human expertise creates bottlenecks, particularly when customizing machines for different clients or scaling operations. These challenges highlight the need for methods that can accelerate code development without compromising reliability, forming the basis for exploring AI-driven solutions.

1.1.2 The Significance of Structured Text

Among the IEC 61131-3 standard programming languages, Structured Text has emerged as a critical tool for modern industrial control. ST is a high-level language which is similar to Pascal in syntax and semantics. It has standard programming features like conditional statements, loops, functions, and data structures. It is thus especially well-suited to the execution of more complex control logic, mathematical calculations, and algorithmic manipulation that are awkwardly described in graphical languages such as Ladder Diagram.

The increasing complexity of the industrial processes has required more sophisticated control schemes than the simple Boolean logic. An example of this is the use of Proportional-Integral-Derivative (PID) controllers, signal filtering, real-time diagnostics, interlock mechanisms, safety functions, and advanced state machines in the control applications. ST is more efficient and

readable method of control programming in such cases.

In addition, ST enables modular and reusable codes to be developed, which is significant in handling large-scale automation projects. Functions, function blocks, and user-defined data types are some of the features that aid maintainability and scalability. This has made it an inevitable language to the modern automation engineers particularly in manufacturing and packaging industries.

ST is extensively used at Coesia in machines where integration of several sub systems should be coordinated precisely, e.g. time, speed, and error management is essential in packaging lines. Its advanced nature enables the engineers to use reusable and modular codes, which enhances maintainability of projects. However, ST programs also create complexity, which adds additional pressure on engineers. It is difficult to write and validate hundreds of lines of code for a single machine, which can be time-consuming and error-prone.

All these benefits notwithstanding, ST programming is a niche skill, frequently demanding expertise in that specific field and the knowledge of industrial control systems. ST code development, testing, and deployment is a labor-intensive process and traditionally carried out manually by years-experienced control engineers. This creates a bottleneck regarding scalability, cost, and development time particularly as industrial systems are increasingly becoming interconnected and data driven.

1.1.3 Rise of AI and Large Language Models in Code Generation

In line with the development of industrial automation, recent years have been marked by revolutionary progress in Artificial Intelligence specifically in the area of Natural Language Processing (NLP) and code generation. LLMs, including GPT-3 [20], Codex [23], AlphaCode [68], PaLM [25], Code LLaMA [93], GPT-4o [55], and gpt-oss [6] have shown a great capacity to produce human-like text, answer questions, translate languages, and most importantly, writing source code.

Such models are trained using large corpora of natural language and programming languages, and can thus learn syntax, semantics, and even regularities in common software development practices. As an example, GPT-5-Codex by OpenAI is fine-tuned on millions of publicly available GitHub repositories and supports more than a dozen programming languages, such as Python, JavaScript, C++, and etc. PaLM and Code LLaMA developed by Google and Meta, respectively, have extended even further the possibilities

of LLMs in domain-specific applications [25][93].

Software development has been able to incorporate LLMs into the workflow. Code completion[57], documentation generation [73], debugging, and refactoring are now available to developers through tools such as GitHub Copilot [23], Amazon CodeWhisperer[1] (it is now being integrated into Amazon Q Developer), and TabNine[2]. Such tools save time of development and add productivity, particularly in repetitive or boilerplate code. Moreover, it has been demonstrated that LLMs can perform close to human on programming tasks, like HumanEval [23], MBPP [11], and CodeXGLUE [72].

With this success, there is now interest in pushing the capabilities of LLMs to Domain-specific languagess (DSLs) e.g., Structured Text (manufacturing and packaging solutions). Unlike other programming languages, ST is less represented in open datasets, and industrial projects also often contain proprietary data. This makes training or fine-tuning LLMs for ST a non-trivial challenge.

1.1.4 Challenges in Applying LLMs to Structured Text Code

Although the concept of applying LLMs to help with the process of ST coding is very promising, there are some serious issues that are currently preventing the further development of the field.

- Lack of Public Datasets for ST Code:

  The training data quality and quantity has an immediate effect when it comes to the performance of LLMs. Python, JavaScript, and other languages have a lot of open-source repositories, tutorials, Stack Overflow conversations, and academic standards. Structured Text code, on the contrary, is scarce in publicly available datasets due to the following factors:

  (1) Majority of ST code is proprietary to industrial systems.

  (2) Industrial vendors and integrators consider this code to be intellectual property.

  Therefore, there is minimal ST code on the open source in GitHub and even fewer have been annotated and curated to be used in machine learning. This absence of data gives rise to a crucial bottleneck.

[1] `https://aws.amazon.com/q/developer/`

[2] `https://www.tabnine.com/`

LLMs cannot be trained on syntax, structure, and domain specific patterns to produce valid and meaningful ST code without enough training examples. In addition, the limited ST datasets are usually very heterogeneous with regard to vendor-specific extensions (e.g. Siemens, Allen-Bradley, and Beckoff) and thus not at all generalizable across platforms.

- Absence of ST Benchmarks and Evaluation Protocols:

  In natural language processing and general-purpose programming, standardized benchmarks play a vital role in evaluating and comparing model performance. For example, the HumanEval benchmark [23] tests LLMs ability to solve algorithmic problems with code. However, no such benchmarks currently exist for ST. There are no accepted metrics for evaluating the syntactic correctness, semantic validity, or industrial applicability of AI-generated ST code. This leads to a lack of reproducibility and comparability in research. A generated ST snippet may compile without errors but still be functionally incorrect or unsafe for deployment in a real system. Therefore, any evaluation of LLMs in this domain must go beyond syntax and address deeper issues such as logic correctness, runtime behavior, and compliance with industrial standards.

- Limited Tool Support for Automated Validation:

  The reliability of the code produced by Artificial Intelligence must be ensured by the presence of the tool chains that are able to parse, simulate, and validate the code. ST compilers and simulation environments (e.g., CODESYS and TwinCAT) are all proprietary and in general are even not easy to integrate in machine learning pipelines. There are few open-source alternatives, and vendor APIs tend to have little or no documentation or be restricted.

  This causes a limit in the automation of testing and validation of the generated ST code. An interesting possibility is syntactic validators or compilers in a feedback loop during training or generation (e.g. compiler-in-the-loop). Nevertheless, it involves closing the gap between AI models and conventional industrial software tools, which remain underdeveloped.

1.1.5 Relevance in the Industry 4.0 Context

In this context, efficient ST programming is more important than ever. Companies like Coesia face pressures to:

- Deliver customized machinery quickly.
- Reduce engineering costs while maintaining reliability.
- Scale software solutions across multiple production lines and locations.

By integrating AI-driven code generation with ST programming, it is possible to accelerate development, reduce errors, and improve maintainability, supporting Industry 4.0 objectives. Moreover, developing methods for low-resource LLM deployment ensures that these solutions are not just theoretical, but also practical for real-world industrial environments. However, these achievements are highly dependent on the quality of the data, validation, and evaluation problems mentioned above.

1.2 Problem Statement

In modern industrial settings, automation systems heavily depends on software that determines how machines operate. The main component of this software is the Programmable Logic Controller that is programmed in various languages e.g., Structured Text, specified in IEC 61131-3 standards. Although, ST is highly used in various industries such as manufacturing and packaging, writing and keeping ST code is a manual, domain specific, and time consuming process. Writing this code requires significant time, domain knowledge, and careful validation, especially in companies like Coesia where machines must be adapted to customer-specific requirements.

Meanwhile, recent advancements in LLMs demnostrates the potential to reduce this effort because these models are trained on vast corpora of code and natural language and have shown impressive results in general-purpose programming tasks e.g., automatic code generation, code completion, and code refactoring. However, their application in industrial settings, especially in the generation of ST code, is still very immature. The main reasons are the lack of publicly available datasets, the proprietary nature of industrial projects, and the absence of benchmarks or validation pipelines dedicated to ST. This develops a complex problem:

- Domain mismatch:

  Majority of the current available code LLMs were not trained or exposed to ST code and either doesn't produce or produce broken ST code which perform poorly with syntax, logic consistency, and execution validity.

- Computational demands:

  The demand that these models usually have in terms of compute or energy resources that it takes to be able to run will not typically be feasible at the edge of an industrial setup.

- Licensing and legal restrictions:

  A lot of state-of-the-art LLMs (e.g., LLaMA, GPT-4) are published either with restrictive licenses (non-commercial) or via API, which create the IP, compliance, and deployment risks in the proprietary industrial sector. What is more, the opacity of training dataset sources may cause legal confusion over generated code.

The situation is further exacerbated by the lack of datasets, benchmarks, and tooling for Structured Text in the LLM training and evaluation pipeline. In contrast to popular programming languages (like Python or JavaScript), the ST domain has a limited representation in publicly available datasets, and there are no standard methods of validating or testing the produced PLC code on a variety of industrial platforms.

In Summary, the most existing state-of-the-art general-purpose Large Language Models are inappropriate for ST code generation due to low domain-specific performance, high computational costs, and restricted licensing conditions. The given research addresses this gap by creating an automated pipeline for synthetic Structured Text dataset generation, development, and evaluation of code LLMs that are lightweight, license-compliant, and can generate reliable ST code under limited computational resources.

1.3 Research Motivation

The growing demands to adopt intelligent automation in industrial environments has also increased the need of faster, reliable, and more efficient software development processes. The industrial control systems are dependent on the creation and maintenance of PLC programs which are usually written in domain-specific languages, e.g. Structured Text under the IEC 61131-3 standard. Though PLC programming has evolved over the past decades, the actual process of writing, debugging, and maintaining the PLC programs is still a major bottleneck. Writing such code manually is a tedious, time-consuming, and error-prone. It also requires special skills both in control theory and in software engineering, and these skills are becoming rare because of aging workers and changing industrial needs. Moreover, the growing complexity of modern automation tasks such as edge-analytics, IoT

integration, and adaptive control add further complexity to the traditional programming paradigms. These constraints have led to research into the use of AI-based tools, particularly the use of Large Language Models as a way to revolutionize software development in industrial automation.

The present study arises from the need to address these concerns within the context of Coesia, a leading industrial firm operating worldwide in the packaging and automation market. Software plays a relevant role in the machines marketed by the firm, with a significant part of it being written in Structured Text, a domain-specific language used to program the complex automation provided by their machines. Also, the international work of Coesia covers a wide range of countries and industries, which introduces the requirement to have software development infrastructures that support the intellectual property rights and legal licensing implications. Coesia is also extremely strict in its policies regarding the access of its assets, which include the source code for their software. The presence of the need for a code LLM able to "understand" a language for which existing general-purpose code LLMs have no support for, and the cautious approach to private assets, make the in-house solution particularly attractive.

Recently, the rise of general-purpose and code LLMs has demonstrated a strong ability to produce syntactically correct source code in a wide range of programming languages. Such developments have opened the door to the potential use of LLMs in industrial firms where they may automatically translate the high-level functional descriptions into deployable control logic. This transformative potential is another motivation of this study. The research aims at adaptation, optimization, and implementation of code LLMs, which can produce high-quality ST code that meets the standards and best practices of Coesia. By doing so, code LLMs will not only increase productivity of the developers, but also improve the quality and maintainability of industrial control software.

Manual programming of PLC is full of pitfalls which tend to compromise the quality of code. The usual problems are inconsistent naming, no modular design, no or poor documentation, and small logical errors. These shortcomings may lead to safety hazards and an increase in maintenance efforts which also result delays in projects deadlines and higher costs, and an added burden on engineering teams. Since, LLMs can be fine-tuned on curated datasets, domain knowledge and best practices can be integrated into them, so that the resulting code can be consistent, modular, and readable. LLMs can learn to produce code that is compliant with the industrial standards like IEC 61131-3, thereby minimizing the need for extensive manual review and testing. It directly impacts the reliability and safety of automation systems, which are the primary concerns in industrial settings.

Another motivation to conduct such research lies in the fact that AI-based tools can democratize industrial programming. LLMs have the potential to empower the domain experts, engineers, and even non-programmers to make a meaningful contribution to the software development lifecycle. They can also allow engineers with little programming expertise to make significant contributions in building control logics. This reduces the entry barrier, increases the willingness of more people to participate and enhance innovation by including more perspectives in the development process.

Although these opportunities exist, but there are still some constraints that arise when it comes to the deployment of code LLMs in industrial firms. Most state-of-the-art code LLMs (e.g., GPT-4, Claude, code LLaMA, Star Coder, LLaMA 2) require substantial computational resources including high-end GPUs and large memory footprints. They are not suitable for programmers' laptops, which have limited resources allocated to them. The use of Cloud-based AI services brings about other problems, including latency, security, network dependency, and operating expenses as well. An essential aspect of this study is investigating the development of limited computational and memory-efficient code-generating LLM architectures and methodological pipelines that can be executed on Coesia's commodity hardware without compromising the code quality.

Furthermore, there are also the legal and licensing concerns associated with deploying code LLMs in industrial settings provide another critical layer of motivation. Most cutting edge code LLMs are released under non-commercial or restrictive licenses, making their use in commercial products or in regulated settings difficult. Moreover, the opaque training data used by most commercial LLMs raises concerns regarding intellectual property (IP) violations, data privacy, and regulatory compliance. This is particularly very important in industries (e.g., manufacturing and packaging) that have safety concerns. These reasons prompt a close look at license-compliant, transparent, and auditable LLMs that can be used in the industry. The most promising alternative to the off-the-shelf models are open-source ones or custom-trained ones in a specific domain. However, they need a rigorous validation to prove robust and meet the safety requirements.

In addition to addressing the current industrial challenges, this research has a strategic vision to contribute to the broader evolution of intelligent automation engineering. The motivation of this work is to provide the foundations of the next generation of industrial software tools augmented with AI by developing models and techniques that integrate domain expertise, operate efficiently in constrained environments, and abide by the legal frameworks. Such tools can greatly decrease the time to market of industrial solutions, raise the reliability of operations, and make automation development accessi-

ble to more people. Democratization of automation programming may drive innovation, enhance safety of workers due to decrease of programming errors, and efficient use of available resources.

The convergence of these motivations forms a compelling foundation for this research, which makes it focus on the novel AI approaches that align with the real-world demands of manufacturing and packaging automation firms. In the framework of the present study, the vision is to enable a future for industrial firms where low-resource, legally-compliant, and domain-optimized LLMs become affordable, efficient, and essential facilitators.

1.4 Objectives of the Study

This research is driven by the desire to advancing the practical application of Large Language Models in ST code generation in industrial automation settings, under the contraints of limited computational resources and licensing considerations. In this regard, the following objectives were set:

- Creation of Structured Text Dataset:

  The lack of publicly available, heterogeneous, and domain specefic dataset of ST programming is the key barrier to the application of Large Language Model in industrial automation sector. Unlike popular programming languages such as Python and Java, the significant amount (up to 100GB) of publicly data are avaibale. ST datasets are sparse, fragmented, or proprietary, limiting the ability to train or fine-tune LLMs effectively for this domain.

  The study aims at compiling a high quality ST dataset that covers a wide range of industrial automation scenarios, programming constructs, and control logic patterns. The dataset will be carefully curated and annotated to ensure it reflects real world PLC programming practices and adheres to the IEC 61131-3 standard. This dataset will serve as a valuable resource for training, testing, and benchmarking LLMs to develop comprehensive understanding of the syntax and semantics of ST.

- Evaluate existing code LLMs and fine-tuned models:

  The results of performance and robustness assessment of existing code LLMs when processing Structured Text code generation tasks.

- Methods for low computational fine-tuning and domain adaptation:

Industrial environments often impose stringent computational and hardware resource limitations so large-scale LLMs are either uneconomical or not feasible to use especially in on-device or edge inference use cases. The study throws light into the development of fine-tuning techniques and domain adaptation strategies that would minimize the computational resources without compromsing the quality of generated code. These methods include Low-Rank Adaptation, PEFT, and quantization techniques adapted to Structured Text. The research will focus on optimizing model size, inference latency, and resource consumptions to enable deployment of LLM-based coding assistants in industrial control systems supporting on-premises, or edge-device deployment. All these contributions fill a gap in the existing research and industry environments, providing the basis upon which more precise and reliable code generation systems can be developed with specific focus on automation in industries.

- To address licensing and compliance challenges in deploying LLMs for industrial automation:

  In addition to technical performance, this study acknowledges that legal, licensing, and intellectual property are also of great significance in industrial implementation of AI models. The licenses of many advanced LLMs are restrictive, and they might not allow using them commercially or integrating them into proprietary systems. Also, it is not clear where the data about training came from, which also introduces the possibility of copyright violation or compliance with the regulations. The purpose of the study is to examine the licensing environment of LLMs concerning code generation, define the legal limitations that currently affect the industrial application, and suggest the guidelines or frameworks to choose or create license compliant, transparent, and auditable LLM solutions. This study will contribute to safer and more sustainable AI adoption in industrial software engineering by integrating these considerations into the model development and evaluation pipeline.

The contextualization of these applications will assist in situating the overall implications and future of the AI-driven automation programming, which will be of great relevance to the researchers and the practitioners in the industry. Altogether, these aims create a coherent plan to move forward with the application of Large Language Models to Structured Text code generation in practice in the industrial world, between technical possibilities, practical limits and legal requirements.

1.5 Research Questions

This study is based on the following research questions. These questions are formulated to address key challenges and opportunities at the intersection of Large Language Models, Structured Text programming, and Industrial and packaging solutions, particularly under constraints of limited computational resources and licensing considerations.

- How can a high-quality and diverse dataset for Structured Text code generation be constructed to effectively train and evaluate Large Language Models?

  There is a significant bottleneck to empower Large Language Models being able to produce valid and semantically meaningful ST code, which is due to the lack of publicly available, well-marked, and domain-specific datasets. The question under investigation is what a representative ST dataset would include, e.g., data sources could be industrial control libraries, open-source projects, or synthetic valid data. It also looks at annotation standards, variety of programming constructs, and incorporation of domain-specific scenarios like safety-critical systems, real-time control, and etc. The aim is to establish optimal practices in the generation of synthetic dataset with appropriate trade-offs between quality, size, and usefulness in automation industry.

- How well existing code LLMs perform on ST code generation?

  Code LLMs trained on a wide range of programming languages and natural language corpora demonstrate remarkable code generation capabilities but often fail to meet the strict syntactic and semantics of ST. Model performance is measured in this question with quantitative measurements of syntax error rates, semantic correctness as validated by PLC compilers or some pre-defined metrics like Pass@k. It also aims at pinpointing certain failure modes of LLMs in this field and the influence of fine-tuning on ST-specific data on these results.

- What techniques can enable efficient fine-tuning and domain adaptation of LLMs under limited computational resource constraints typical of industrial environments?

  A lot of industrial organizations do not have high-performance computing infrastructure that is needed to train, fine-tune, or infer large-scale code LLMs. This question is devoted to the discussion of Parameter-Efficient Fine-Tuning techniques like LoRA and quantization techniques. The aim is to create a pipeline, which would be able to trade-off between

the computational efficiency and code generation quality, enabling deployment of LLM-powered code assistants on premises end-user devices or PLC-based systems.

- What is the effect of licensing, intellectual property, and compliance in the use of LLMs in industrial automation software development?

  Proprietary or open-source licensed LLMs come with complicated licensing conditions that could limit the commercial use or the integration into the industrial settings. The question will address the legal landscape of AI-based models focusing on how licensing impacts the selection, deployment, and customization of LLMs for industrial settings. It also consider data provenance concerns, ethical implications of training data copyright, and regulatory compliance requirements of industrial safety and security standards as well.

1.6 Scope of the Study

The scope of this research is defined to provide clarity and relevancy while addressing critical aspects of applying Large Language Models to Structured Text programming within industrial settings.

- Programming Language Focus:

  This study only focuses on ST, which is one of the five languages specified in the IEC 61131-3 standard of PLC. Ladder Diagram, Function Block Diagram, Sequential Function Charts, or Instruction List are other languages out of scope. Such a narrow implementation can be used to explore in-depth the unique syntax and semantics of ST, which are challenging to LLM-based code generation.

- Industrial Automation Domain:

  The application scenario is in industrial automation systems in which PLCs are used to control manufacturing and packaging solution. The study focuses use-cases icluding safety-critical, control logics, and real-time that demand reliability and maintainability in the code generated by LLM.

- Model Types:

  The study is focused on LLM based models as they have a good code generation abilities. It contains the pre-trained open sourced code LLMs.

- Computational Limitations:

  Since the study is carried out in the context of industrial settings and has resource limitation constraints. The study prioritizes low-resource fine-tuning methods, model quantization, and efficient inference. It includes exploring lightweight architectures that can be deployed on edge or on-premises devices instead of relying solely on Cloud-based services.

- Dataset Scope:

  In this study, the focus is to create a synthetic ST dataset. The code snippets on which the dataset developed or used in this research is based on a wide spectrum, including simple code constructs (variable declaration, control flow), to complicated automation functions (interlocks, PID control, communication protocols). The data is supposed to represent real-world diversity without proprietary or confidential industrial code unless the code is anonymized.

- Licensing and Legal Issues:

  The study presents an examination of licensing conditions of mainstream LLMs and how they affect their adoption in industry, though it is not a legal advice. Companies that incorporate the use of AI should consult professionals to avoid the trap of contractual and intellectual property problems.

- Use Case Limitations:

  The primary focus is on code generation tasks such as automatic ST code synthesis from natural language or formal specifications or code completion. Other activities in software engineering such as testing, deployment, integration, or user experience design are not covered.

1.7 Limitations

Although this research intends to be of great contribution, but still it has a number of limitations.

- Dataset Representativeness:

  Although a diverse and high-quality ST dataset will be created, it might not be able to take domain-specific nuances or proprietary control logic employed in specialized industries. Limited generalizability can also be

a factor that restricts the availability of the dataset because of confidentiality agreements.

- Model Performance Boundaries:

  The focus is on low-resource fine-tuning of code LLMs. The adapted models may not be able to handle some complex programming tasks.

- Licensing Complexity:

  Although this study offers some directions on licensing and compliance, these legal issues are very situational and dependent on jurisdictions. Organizations have to take it upon themselves to ensure that there is adherence to the law and to the contracts.

- Industrial Validation:

  The successful implementation and industrial demonstration of LLM-generated ST code in actual live industrial control systems are outside the scope of this thesis. Areas to be improved in future work include long-term deployment, user acceptance, and safety certification.

- Ethical and Security Issues:

  The ethical concerns of AI, data privacy, cybersecurity threats, and unintended effects of automated code generation are mentioned, yet they are not discussed in any detail, which should be considered a valuable research topic.

1.8 Thesis Structure

This dissertation is organized as follows:

- **Chapter 2** provides a comprehensive literature review on Large Language Models, code generation techniques, and Structured Text (ST) programming.
- **Chapter 3** presents the Industrial Feasibility of PEFTpyCoder: A Parameter-Efficient Fine-Tuned Model for In-house Code Assitance.
- **Chapter 4** details the pipeline for synthetic ST dataset creation and describes the implementation process.
- **Chapter 5** addresses licensing and innovation management aspects related to industrial adoption of AI-driven code generation tools.

- **Chapter 6** presents the experimental results and provides a detailed discussion on the performance of the proposed approaches.
- **Chapter 7** concludes the dissertation by summarizing key findings and highlights potential directions for future research.

Chapter 2

Literature Review

The rapid growth of Large Language Models have triggered a lot of academic and industrial attention, especially in the context of their possible effects on code generation and automation activities. This chapter reviews the existing literature in a focused and thematic manner, emphasizing work that is most relevant to industrial automation and domain-specific programming languages such as Structured Text (ST), and code LLMs. Selection prioritized studies that illustrate the evolution of natural language-to-code techniques, the design and evaluation of code-focused LLMs, and approaches that consider resource efficiency and practical deployment constraints.

2.1 Overview of Large Language Models

Large Language Models represent a major breakthrough in Artificial Intelligence, especially in natural language processing (NLP) and Machine Learning (ML). These models are built on the Transformer architecture initially developed by Vaswani et al. [111] in 2017 and which completely transformed the way how machines understand and generate human language. Since then, LLMs have served as the basis of development of numerous applications from chatbots and virtual assistants to automated coding tools.

The main idea of LLMs is to train models on large corpora of text to learn the statistical dependencies between words and phrases. Transformer based models apply self-attention to assign relevance to different words in a sequence, enabling them to comprehend context more effectively than earlier models such as Recurrent Neural Network (RNN) and Long-Short-Term Memory (LSTM). This allows the LLMs to produce contextually correct coherent text.

One of the earliest examples of LLMs are BERT (Bidirectional Encoder

Representations from Transformers) and GPT (Generative Pre-trained Transformer).Although BERT[34] is mainly applied to understanding tasks such as classification and question answering, GPT models are designed for generative tasks, making them better suited for applications involving code generation. GPT-2[87] and GPT-3[20], published by OpenAI, received significant attention to LLMs because of the quality of generating coherent text based on a single prompt.

GPT-3, particularly its 175 billion parameters model marked a turning point in language modeling because it showed emergent capabilities like zero-shot and few-shot learning. These abilities enabled models to address tasks without being trained on them, merely by reading instructions and examples in natural language. The next model, GPT-4 [4], continued to show even better performance in several areas, such as programming, mathematics, and logical reasoning.

Code generating LLMs such as OpenAI Codex[72], CodeBERT[37], AlphaCode[68], and CodeGen[80] use the same architecture and are trained on corpus of source code. Such models show exceptional results in translating natural language into executable code, something that is historically regarded as a challenging task as it requires both syntactic and semantic accuracy.

LLMs require a lot of computations even though they are potentially powerful. To give an example, GPT-4, with approximately 1.8 trillion parameters, requires around 25,000 A100 GPUs for its training [119]. LLMs require large amounts of memory and processing power even when they are in inference mode, and thus LLMs are difficult to implement on low-resource systems.

2.2 Evolution of NL-Code Techniques

The process of translating natural language (NL) to code, sometimes called NL2Code, has evolved significantly over the past two decades. Initially, rule-based systems and expert-systems dominated the field. These methods were based mainly on hand-written rules and Domain-specific languages which made them limited in terms of scalability and generalization to other tasks.

Early approaches involved heuristic methods, probabilistic grammar-based techniques, and dependency parsing. Although these techniques helpful in confined settings but did not generalize to other programming paradigms or allow one to deal with the ambiguity of natural language. The Hidden Markov Models (HMM) and n-gram statistical models proposed by [79] and [88] were the first step towards automatic code completion and synthesis. However, these models were weak when it came to long-term dependency

and semantics.

The paradigm was changed with the introduction of deep learning. The NL2Code task started to be addressed using Convolutional Neural Networks (CNNs), RNNs, and LSTMs that provided superior performance as they were data-driven. However, such models required a large number of annotated pairs of NL and code that were not always easy to find. In addition, their architecture struggled to capture long-range dependencies in complex sentences or large blocks of code.

A significant breakthrough came with the adoption of Transformer-based models for NL2Code. Shah et al. [96] managed to apply the Transformer architecture to Python code generation based on the natural language description in 2021. The ability of this architecture to model context and relationships over long sequences made it particularly well-suited to the complexities of both human and programming languages.

Subsequent advancements led to the release of models pre-trained on massive code corpora, enabling zero-shot and few-shot code generation. Codex [23] was an early model, which was trained on GPT-3 and had shown to be able to solve actual programming problems from natural language instructions. Likewise, CodeBERT [37] was an extension of BERT architecture to code summarization and code generation.

Modern NL2Code systems are more complex, and apply instruction tuning, reinforcement learning with human feedback [83], and chain-of-thought prompting in order to enhance the quality and relevance of generated code. These techniques are bringing the limits of what can be done in software automation and creating new opportunities in programming by non-experts and accelerating development processes.

However, there are still some difficulties. NL2Code models can be very weak at handling ambiguous inputs, lengthy code generation, and domain specific programming languages (e.g. Structured Text). Furthermore, these models are neither easily accessible nor deployable in limited environments due to their dependence on large computational resources.

2.3 Code Large Language Models

Large Language Models have quickly changed the domain of code generation, allowing one to use natural language as a source of the source code. Based on transformer architecture, these models have been trained on large corpora of programming and natural language data and can be used to do tasks such as predicting API calls or solving competitive programming problems. With the increasing need for software automation, code-generating LLMs are being

considered not only because of their potential for increasing productivity but also because of their ability to democratize programming. This section reviews the evolution, capabilities, and limitations of prominent LLMs for code generation, focusing on their architectural choices, training data, and evaluation benchmarks.

Li et al. [68] introduced AlphaCode, which produced the code on a similar level to the competitive programmers. In contrast to the previous models, which only succeeded on trivial tasks, AlphaCode approached more complex, algorithmically diverse tasks, provided by one of the most popular platforms of programming competitions Codeforces. It is based on a large transformer-based encoder-decoder architecture and trained in two stages; 715 GB of open-source GitHub code is used in pre-training and a curated dataset named 'CodeContests' is used in fine-tuning, which greatly minimizes the number of false positives due to the use of rigorous test augmentation. AlphaCode employed massive sampling (at least 1 million candidate programs per problem) and eliminated candidates through example-based testing and clustering, only the few high-quality programs were submitted. AlphaCode had a place among the top 54.3 % of human competitors in 10 Codeforces competitions during evaluations, which was an indication of strong algorithmic reasoning abilities.

Thoppilan et al. [107] introduced LaMDA, a family of dialog-optimized Transformer-based language models in a size of up to 137B parameters and trained on 1.56 trillion words of web and dialogue data. Although scaling of the model enhanced quality of output but it did not impact much on safety and truthfulness. To overcome this, LaMDA used fine-tuning together with annotated data and integration of external tools (e.g. retrieval systems) to enhance groundedness and safety. The authors proposed to use structured evaluation metrics, quality, and safety to assess the LLMs in other areas such as code generation. The architecture, training strategy, and deployment aspects of LaMDA pointed out general issues and methods within the development of trustworthy, domain-adaptable language models.

Chowdhery et al. [25] presented PaLM (Pathways Language Model) a very large-scale decoder-only Transformer model that has been scaled to 540 billion parameters and trained on the Google Pathways system that allowed efficient parallel training. The architecture adhered to the typical Transformer decoder architecture and major advancements included parallelism, optimization, and mixed precision training as a way of keeping the computational scale down. PaLM was trained on 780 billion token corpora consisted of a combination of filtered web documents, books, Wikipedia, and source code (GitHub), and heavily biased towards high-quality data. Such a diversed collection of data allowed PaLM to complete a variety of tasks, such as rea-

soning, code generation, and translation. The model was tested on more than 29 NLP tasks in areas such as question answering, natural language inference, reading comprehension, and etc. Interestingly, PaLM achieved state-of-the-art performance on various benchmarks, including BIG-bench, SuperGLUE, and MMLU. Despite having remarkable achievements, the paper has certain limitations as the high computational cost, the difficulty of achieving factual accuracy, and the ethical issues of bias and toxicity in large-scale models. Also, in some cases, domain-specific fine-tuning or integration of external tools still beats scaling at particular tasks, despite the resulting general capability. Overall, PaLM was an important leap forward in LLMs, providing evidence that architectural scaling and high-quality data can bring a general and robust performance, yet pointing to the need to study safety, interpretability, and deployment issues on a large scale.

Christopoulou et al. [26] proposed PANGU-CODER, a decoder-only language model pre-trained on the task such as program synthesis, i.e. given a natural language description, the model can generate a program that implements the same functionality. It was based on the PANGU-alpha architecture, uni-directional transformer with an added query layer, which was intended to produce functionally correct Python code. The training was performed in a two-stage plan. In the initial phase the model was trained end-to-end with Causal Language Modeling (CLM) over a large corpus of raw Python code with docstrings and inline comments where possible. At this unsupervised stage, the model acquired the general architecture and syntax of programming languages. The second step was supervised training of aligned natural language-code paired with CLM and Masked Language Modeling (MLM) objectives and thus reinforced the model to produce code when given natural language input. It was evaluated against the HumanEval and MBPP benchmarks that test the functional correctness of produced code through the execution of unit tests. Findings indicated that model performed competitively when compared to the state-of-the-art models, like Codex, trained on smaller context windows and with fewer data. Additional performance was achieved by fine-tuning, making a version known as PANGU-CODER-FT. It was trained on competitive programming problem and continuous integration system data, and achieved better pass@k rates, especially on MBPP benchmark. The authors also examined the possibilities of improvement by filtering after generation (e.g. unit test verification, type checking) and offered a data selection technique by utilizing few-shot similarity with CodeBERT embeddings. These methods allowed performing domain adaptation efficiently and enhanced the quality of code generation without a large growth in computational cost.

Shen et al. [97] proposed PanGu-Coder2 an open-source code generation

large language model, based on the StarCoder model with 15B parameters. The authors proposed a new training framework named RRTF (Rank Responses to align Test & Teacher Feedback) to increase the efficiency of code generation. In contrast to the classical approaches of reinforcement learning where the reward model was usually complexed and needed high computational power, RRTF initiated a more effective and scalable approach where the outputs of the model are ranked according to the unit test results and teacher model instructions. The training was done by choosing high quality responses based on test feedback and heuristics, and by training the model to prefer these ranked outputs. Using the same methodology the PanGu-Coder2 was trained on 68,000 instruction-solution pairs that were generated through the Evol-Instruct technique, which guaranteed quality and diversity of the data. PanGu-Coder2 set a new state-of-the-art results of the open-source models, reaching a pass@1 accuracy of 62.20% on the HumanEval benchmark, surpassing the previous best models, such as WizardCoder and StarCoder. It also showed better results on the CoderEval and LeetCode benchmarks, which illustrated high generalization to coding tasks in the real world. Besides, the model was compatible with a range of quantization methods, which allowed saving a significant amount of memory and inference time without a significant loss of performance. The paper concluded that ranking-based feedback is a feasible and efficient alternative to reinforcement learning to align code LLMs with desirable behaviors, which opened the path to future enhancement of open-source code generation systems.

Meta AI released Code Llama [93], a family of Large Language Models based on Llama 2 and explicitly fine-tuned to perform programming tasks. They proposed models like 7B, 13B, 34B, and 70B parameters as general-purpose (Code Llama), Python-specialized (Code Llama-Python), and instruction-following (Code Llama-Instruct) variants. The features included in these models were infilling, processing of long-context (up to 100,000 tokens), and instruction fine-tuning to make output safer and more useful. The domain-specific model beat bigger general models, like Llama 2 70B, on HumanEval and MBPP benchmarks proved the effectiveness of domain-specific training. Code Llama models have state-of-the-art performance across open models on several benchmarks, such as HumanEval, MBPP, APPS, and MultiPL-E, through intensive tests. Code Llama-Instruct variant improved alignment and safety further by combining instruction data and a self-instruct dataset, created by prompting and executing the model and capturing the feedback. This paper key featured the importance of model specialization, long-context fine-tuning and alignment methods in the development of open-source code generation capabilities.

Artetxe et al. [10] introduced a compute-efficient way to scale the train-

ing of Large Language Models with Mixture of Experts (MoE). As opposed to dense models, which activate all parameters on every input, MoE models only activated a small subset of parameters, usually two out of thousands of experts, on each token, and therefore MoE models scaled the capacity of the model with much less proportional increase in computational cost. The authors use top-2 gating mechanism, load balancing techniques, normalization strategies to maintain stability, and good use of experts during training. Their performance demonstrated that MoE models were more efficient in terms of performance per FLOP than dense models in a wide range of tasks such as language modeling and machine translation. This paper demonstrated the opportunity of scale language models efficiently with sparse architecture and preserve or enhance quality, and it is a promising future development path of large-scale natural language processing.

Fried et al. [38] proposed InCoder, the generative Transformer-based language model used for both code synthesis and code infilling, accommodating a vast spectrum of code generation and editing via masking and infilling in a single unified architecture. InCoder is trained differently, with a causal masking objective, compared with conventional autoregressive code models that only supported left-to-right generation, and thus abled to condition on both previous and subsequent tokens, enabling arbitrary infill, such training scheme allowed zero-shot generalizing among an array of code editing tasks like type annotation, docstring prediction, variable renaming, multi-line insertion, and training without task-specific fine-tuning. The 6.7B parameter model were trained using a big corpus of permissively licensed code and Stack Overflow sources. The evaluation results indicated that InCoder not only demonstrated good performance on common synthesis benchmarks such as HumanEval but also significantly surpassed left-to-right baselines on new infilling benchmarks. InCoder offered a versatile basis of real-life software development tools by merging the capability of both synthesis and bidirectional editing in a single model.

Zan et al. [123] proposed CERT (Continual Pre-Training on Sketches) framework which is a new library focused on code generation method that did not require expensive paired text-code data. Rather than relying on a supervised fine-tuning process, CERT leveraged the insight that code snippets using third-party libraries (e.g., Pandas, NumPy) tend to exhibit similar structural patterns, or sketches. The framework broke down the generation of code into two phases, a sketcher that predicted a skeleton of the abstract code by anonymizing user-defined parts, and a generator that filled the details to generate executable code. The approach was also computationally efficient because both modules were trained on unlabeled code corpora in a continuous fashion with a lightweight base model (e.g. PYCODEGPT

with 110M parameters). The authors also proposed two new benchmarks; PandasEval and NumpyEval that consisted of real-world programming problems. As experimental findings indicated that CERT was far better than current models (including larger ones, such as Codex and GPT-3) at producing correct library-based code, especially on tasks that involved multiple API calls. This work highlighted the importance of structure-aware pre-training in domain-specific code generation under resource constraints.

Clement et al. [27] proposed PYMT5, a significant new development in this area, that allowed multi-modal translation between Python method signatures, docstrings, and method bodies with a single unified model. In contrast to the prior methods that referred to the code and the language individually or regarded the one-way translation, PYMT5 was based on a text-to-text model based on T5, enabled it to generate code based on natural language descriptions and vice versa. The model was trained on the largest parallel corpus of Python methods and docstrings ever (more than 26 million methods and 7.7 million docstring pairs), and it can do style conditioned docstring generation produced state-of-the-art results in terms of BLEU and ROUGE scores over GPT-2 baselines. Multi-mode training strategy guaranteed efficient learning of incomplete data using all possible combinations of method features, and its pre-training strategy made convergence much faster. PYMT5 thus demonstrated the effectiveness of large-scale, multi-task pretraining for automated code generation, code documentation, and comprehension which made it highly relevant for the context of modern software development environments.

Gunasekar et al. [43] presented phi-1, a 1.3B parameter Transformer-based code generation language model that attained state-of-the-art results (HumanEval: 50.6%; MBPP: 55.5%) on benchmarks such as HumanEval and MBPP. The main lesson of the work was that the quality of training data could make up the scale. The authors did not use any massive web-scale corpora and instead constructed a smaller, cleaner, and more textbook-like dataset that consisted of filtered educational code snippets and synthetic Python textbooks and exercises that were built with GPT-3.5. The model was only trained on eight A100 GPUs in four days and fine-tuned for seven hours. Amazingly, phi-1 is much better than even much larger models (e.g. StarCoder, Replit) and even has emergent capabilities like accessing external libraries (e.g. PyGame, Tkinter) and solving logical reasoning problems, despite not being directly trained on those. The authors also confirmed that the performance of phi-1 is not caused by data contamination through such methods as decontamination and custom evaluation tasks. This paper questions the traditionally accepted emphasis on model and data scale with its evidence that a properly designed, didactic data, like a good textbook,

can help small models to perform well.

Phi-1.5 is a 1.3B parameter model constructed by Microsoft Research [67] that is at least as good as, and in some cases better than, significantly larger models on a number of benchmarks. Phi-1.5 was comparable to much larger models, including LLaMA-2-7B, Vicuna-13B on reasoning datasets, including WinoGrande, ARC-Challenge, and SIQA, and was significantly better than other models of the same scale. phi-1.5 posed a challenge to the standards of language knowledge and understanding such as PIQA, HellaSwag, MMLU, OpenbookQA, SQuAD, and performed better than models such as Falcon-RW-1.3B and GPT2-XL. In particular, phi-1.5 beat the models of equal or greater size in elementary mathematics (GSM8K) and coding (HumanEval, MBPP) comprehensively and even on human assessment of code generation using Python-based code generation HumanEval. It accomplished this performance and was much more efficient, took only 1,500 GPU-hours to train. The inference latency per token was less than 3ms, and the training dataset consisted on 150 billion tokens, which was much smaller compared to terabyte-level datasets of larger models. These findings demonstrated the efficiency and effectiveness of phi-1.5, which achieved due to its utilization of the high-quality synthetic training data that employed the so-called textbook-like training data instead of large amounts of the web-scraped text.

Chandel et al. [22] introduced JuPyT5, a transformer-based model capable of assisting with data science in Jupyter notebooks. The authors introduced a new benchmark, Data Science Problems (DSP) consisting of 1,119 curated problems that were grouped into 306 pedagogical notebooks, each with natural language descriptions of the problem, context cells, and assert-based unit tests. JuPyT5 trained on a giant corpus (more than 7 million distinct Jupyter notebooks) on GitHub with a code-infilling task to fill in missing cells of notebooks with neighboring ones, and control tokens that specified the type of cell. With 100 sampling tries, the model was able to pass the DSP benchmark with a rate of 77.5%, and its performance significantly enhanced when provided unit tests and additional context cells. It was interesting because JuPyT5 with few parameters e.g, 300M was better on MBPP benchmark than larger models (Codex), but not on HumanEval. These findings pointed out that smaller models that were domain-specific could be competitive with much larger general-purpose models. In the paper, it is concluded that JuPyT5 demonstrated that effective data science assistants could be built and integrated into notebook environments, which can be used in education and practice of data science.

Svyatkovskiy et al. [104] presented IntelliCode Compose, a general purpose multilingual code completion tool developed by Microsoft, which could be used to assist developers that would give them complete lines of syntac-

tically correct code in an extensive range of programming languages, such as Python, C#, JavaScript, and TypeScript. It was based on a proprietary transformer model of GPT-2 style called GPT-C, trained with 1.2 billion lines of code and was superior to the typical code completion tools as it predicted the sequence of any type of tokens, including method names, variables, arguments, and language keywords. Performance of the system was much better in average edit similarity of 86.7% and perplexity of 1.82 in Python. It employed client-side caching, efficient beam search decoding, Cloud delivery in its real-time suggestions, and a less than 100ms response time. The authors also commented on multilingual modeling (MultiGPT-C), knowledge distillation based compression of models, and how to reduce the risk of security breaches of sensitive information. On the whole, IntelliCode Compose was an important milestone in the use of AI to solve software development problems by making code generation in IDEs such as the Visual Studio Code smarter and faster.

Husain et al. [56] introduced the CodeSearchNet, a benchmark and dataset that could be used to develop the task of semantic code search, which involved querying natural language to retrieve relevant pieces of code. It has a huge set of about 6 million functions in 6 different programming languages, of which about 2 million were accompanied by documentation as training data. The CodeSearchNet Challenge consisted of 99 real world natural language queries with expert annotations of relevance scores of candidate code snippets to facilitate strong evaluation. The paper described the process of data collection, cleaning and annotation, and also provided several baseline models, both conventional ones, like the traditional ElasticSearch, and neural ones, like Bag-of-Words, CNNs, RNNs, and Self-Attention architectures. Surprisingly, the more basic models like ElasticSearch and Neural Bag-of-Words were competitive, which pointed to the issues like ambiguity of queries, poor quality of code, and the absence of context. The authors stated that more advanced models that support code semantics were required and outline this issue as the basis of future studies on code search and similar tasks.

Lu et al. [72] developed CodeXGLUE, a broad benchmark that used to conduct machine learning research on code intelligence. It consisted of 14 datasets that targeted 10 different tasks including code completion, code summarization, code translation, defect detection, code repair, natural language code search, and etc. These tasks involved different programming languages including Python, Java, and C# and were targeted at both code understanding and code generation. The benchmark also offered 3 baseline models CodeBERT (BERT-style), CodeGPT (GPT-style), and an Encoder-Decoder architecture to support model development and evaluation. CodeXGLUE allowed fair comparisons and tracking of progress across the entire area, and

has standardized datasets, baseline systems, evaluation schemes, and considered a Glue-like benchmark on code-based machine learning tasks.

Bavarian et al. [12] introduced a simple yet effective solution to make autoregressive language models performed fill-in-the-middle (FIM) tasks where the model required to generate text based on both left and right context (i.e., both directions). This was done by a process of data transformation, which randomly picked an intermediate range of text, attached it to the end of the input, and marked each part with sentinel tokens. The study revealed that the initial left-to-right generative capacity of learning models on a mixture of regular and FIM-transformed data was not impacted, or what they termed as FIM-for-free. The authors concluded the best practices of FIM training based on numerous experiments on different models of different sizes and in different domains: character-level span selection, context-level transformations, and high FIM rate (up to 90%). They also proposed new standards to better assessed the infill performance and demonstrated that although the FIM can be trained efficiently during pretraining, it took much more compute to train in finetuning. The authors suggested that FIM training was to be a common practice in pretraining autoregressive models because of its effectiveness and practicality.

Luo et al. [74] proposed WizardCoder, a family of code models, which were augmented with a distinct instruction tuning protocol identified as Code Evol-Instruct. The approach used the domain specific heuristics to scale up the simple code prompts with progressively more complex tasks, such as addition of adversarial examples and addition of computation complexity. WizardCoder models with such fine-tuned instructions showed state-of-the-art perform on common benchmarks such as HumanEval, HumanEval+, MBPP, DS-1000, or MultiPL-E. WizardCoder-15B outperformed closed-source models like Claude and Bard, while WizardCoder-34B matched or surpassed GPT-3.5 (ChatGPT), emphasized the critical role of instruction complexity in boosting Code LLM performance.

Chen et al. [23] presented Codex, a fine-tuned GPT model on publicly available Python code on GitHub. The primary goal of this work was to assess the potential of such models to generate functionally correct Python code on the basis of natural language specifications, namely, docstrings. In order to systematically assess the performance, the authors created a benchmark dataset called 'HumanEval', which included 164 hand-written problems with associated unit tests, which made it possible to automatically check the correctness of the code. Pass@k was the primary metric to assess the quality of generated code, and this measured the probability that at least one or more of 'k' samples generated will pass all the test cases. The accuracy of Codex-12B is 28.8% (pass@1) and 72.3% (pass@100) relative to the ear-

lier models which comprised GPT-3 and GPT-J. The paper also presented Codex-S, an improved version that was handled with the assistance of standalone functions that learnt in competitive programming competitions and continuous integration pipelines. Codex-S performed better in terms of the different sizes of modeles, especially when repeated sampling was involved. The reverse task of creating docstrings given code, through a variant called Codex-D, was also another contribution, though this task was possible but much more variable and less accurate than the original task because of the subjectivity and quality of real-world docstrings. Notably, the authors also identified some of the major shortcomings of existing code generation models. Codex had a problem with long chains of operations, variable binding, and abstract thinking prompts.

Le et al. [62] introduced CodeRL, a code generation model with Pretrained Models and Deep Reinforcement Learning suggested a new approach to the reinforcement learning (RL) technique that enhanced the program synthesis procedure using natural language specification. The existing code generation models only used supervised fine-tuning with ground-truth code and did not use other potentially valuable signals like unit test results, which might constraint them on more challenging tasks. CodeRL came in by defining this as an actor-critic RL setting, and training a pretrained language model (CodeT5) as the actor and a critic network that predicted the quality of the generated code in terms of functional correctness using unit tests as feedback. The critic provided token-level rewards which were rich and inform the actor during the fine-tuning process and motivate production of semantically correct code. They also proposed a process of Critic Sampling in the framework, which optimized programs that passed an initial test and fixed programs that failed, through critic guided sampling and model of program repair. The experimental results showed highly competitive outcomes and state-of-the-art performance on the APPS benchmark, and zero-shot transfer to the MBPP benchmark, even at the gigantic scale, e.g. GPT-137B. This paper explained why it was beneficial to incorporate RL and functional feedback both during training and during inference so that code generating models became more correct and general.

BigScience Workshop. [116] introduced BLOOM, an open-source 176B parameter multilingual language model trained by a large-scale (more than 1 thousand participants) community-driven project. BLOOM was pre-trained on ROOTS, a huge multi-lingual corpus with 1.61 terabytes in 46 natural languages, and 13 programing languages. BLOOM was the initiative that focused on making big language models more democratic, that's why the model, and its code released publicly, as there was a certain dominance of proprietary LLMs at the moment. The Megatron-DeepSpeed framework was

used to train the model that consumed 3.5 months of 384 A100 GPUs on the Jean Zay supercomputer in France. To have the most comprehensive possible language coverage and also to have efficient and fair language free tokenization, a Byte-Pair Encoding tokenizer with special tuning on 250,000 tokens were created. BLOOM was trained against a multilingual prompt pool (xP3) and using it on the tasks within the framework of a zero-shot scenario meant that it produced a model BLOOMZ, which showed good performance. The project had a healthy ethical charter that was based on transparency, inclusivity, multilingual equity, and accountability of data governance. BLOOM was not just a new language model, but also became the first large AI system to be created and released in a transparent and collaborative way that was socially responsible.

Nijkamp et al. [80] introduced the family of open-source LLM named CodeGen, focused on program synthesis, namely, multi-turn code generation. They possessed 350 million to 16.1 billion parameters, and they have been trained in stages on natural language (The Pile), multi-language code (BigQuery), and Python specific corpora (BigPython). In contrast to the single-turn paradigm that was available before, CodeGen enabled a multi-turn program synthesis paradigm whereby the user could gradually specify programming tasks with the help of sequential questions in natural language and thus produced more precise and context-aware code. To evaluate, the authors proposed the Multi-Turn Programming Benchmark (MTPB) that was a new set of 115 tasks where multi-step code synthesis was involved. CodeGen models performed better than the earlier open-source models such as GPT-Neo and GPT-J as well as proprietary models such as OpenAI Codex on single-turn and multi-turn significantly. Interestingly, the multi-turn prompting decreased the prompt perplexity and enhanced the program correctness by more than 10% in comparison to the comparable single-turn specifications. The authors also published their training framework library (JAXFORMER) and model checkpoints.

Nijkamp et al. [81] introduced CodeGen2, that seek to make the training of Large Language Models more efficient in terms of programming and natural languages. The author tried to consolidate the main elements of the model architecture, learning goals, sampling methods, and distributions of data into one training recipe to minimize the cost and complexity. On the basis of a large number of empirical tests, they summarized in five main lessons, which were: (1) Prefix-LM was not always superior to standard models that were trained on decoder-based objectives; (2) Infill sampling was not a free feature and could slightly diminish left-to-right generation; (3) a combination of the causal language modeling objective and span corruption objective was effective, which however must not be naive; (4) programming and natural

language data could be combined to provide a multitask capability without much performance loss; and (5) multi-epoch training. However, the paper also provided practical suggestions and published the open-source CodeGen2 family of models (1B to 16B parameters) and training framework to facilitate the future research and advancement.

Wang et al. [112] proposed CodeT5, a unified encoder-decoder Transformer architecture that operated on program code, inoder to address the shortcomings of encoder-only and decoder-only models that have generally been sub-optimal to both tasks e.g., code understanding and code generation. CodeT5 was pretrained using identifier-aware pre-training that enabled the model to explicitly learn to detect and predict identifier tokens (variable and function names) and thereby learnt to capture better semantics in code. The authors also proposed a bimodal dual generation concept to make the model learn code and related natural language comments so that the relation between programming and natural languages could improved. CodeT5 was also capable of multi-task, which implied that it was used to perform most downstream tasks such as code summarization, generation, translation, refinement, defect detection, and clone detection within the same framework. CodeT5 was trained on the CodeSearchNet dataset and additional data. The authors also proposed a new state-of-the-art on the CodeXGLUE benchmark consisting of 14 subtasks, demonstrated increased ability to model both syntax and semantics of code.

Wang et al. [114] proposed CodeT5+, a family of open-source encoder-decoder Large Language Models that employed to accomplish code understanding and code generation. It partly mitigated some of the major weaknesses of current code LLMs, which were more pretrained on fixed models (encoder-only or decoder-only) and less pretrained tasks. CodeT5+ departed by allowing flexible module activation (encoder-only, decoder-only, or encoder-decoder) and training a combination of various pretraining objectives-span denoising, causal language modeling (CLM), contrastive learning, and text-code matching on both unimodal (code-only) and bimodal (text-code) datasets. The authors used pretrained LLMs using shallow encoder, deep decoder architecture to scale efficiently which was far less expensive to train. They also used instruction tuning that utilized to match models with natural language prompts. Wide-scale experiments on 20+ benchmarks demonstrated that CodeT5+ has state-of-the-art performance on code generation, completion, summarization, retrieval, and math programming tasks. The instruction-tuned CodeT5+ 16B model scored better on the HumanEval benchmark than OpenAI code-cushman-001 which scored 35.0% pass@1 and 54.5% pass@10.

Wang et al. [115] proposed CodeT5Mix, a pre-trained language model

for code understanding and code generation. Compared to the traditional models whose architecture were fixed, CodeT5Mix had the possibility of a free combination of the encoder-decoder transformer blocks that could freely added according to the task needs. It had a two-stage pre-trained model which was based on a large set of self-supervised tasks such as span denoising, causal language modeling (CLM), contrastive learning, and text-code matching with the use of code-only and code-text large-scale datasets. CodeT5Mix used task-specific feedforward network (FFN) experts in a shared framework to address task interference while maintaining parameter efficiency. CodeT5Mix achieved state-of-the-art or close to it performance on more than 20 benchmark datasets, a large number of code-related tasks such as text-to-code retrieval, code generation, code summarization, and math programming. Its unified design also allowed it to operate effectively as a retrieval-augmented generator, made it a robust and versatile model for code intelligence applications.

2.4 Overview and Challenges of Structured Text Programming

Structured Text (ST) is a textual, high-level programming language, which is specified in IEC 61131-3 standard used to program industrial automation such as Manufacturing and Packaging solutions. It is a popular programming language in Programmable Logic Controllers, remote terminal units (RTUs), and other embedded industrial controls. ST is readable, maintainable, and incorporates the features of other traditional programming languages such as Pascal and Ada, which makes it appropriate when applying complex logic in Manufacturing and Packaging solutions.

The main strength of ST is that it allows to represent the complicated logic and mathematical calculations in the structured and comprehensible form. In contrast with graphical languages such as Ladder Diagrams or Function Block Diagrams, ST lets the developers write in more familiar syntax, including loops, conditionals, and variable declarations. This makes it especially useful in applications that require advanced control strategies, data handling, or integration with other software systems.

Although ST is strong, it poses special challenges to code generation and automation. The absence of publicly available data is one of the significant problems. The majority of ST code is proprietary and encapsulated in an industrial context, so it is hard to construct and condition models to comprehend or produce it. Also, ST is usually stored in XML tags (e.g., .tcPOU files

in TwinCAT), which introduces another level of complexity to the extraction and preprocessing of the data.

Industrial applications are also deterministic and thus require a high level of precision and accuracy of generated code. Inaccuracies in ST may result in failures in the system, hence automated generation is more risky than general-purpose programming languages. Moreover, ST does not have such a wide community and documentation as languages such as Python or Java, and it is more difficult to apply AI solutions properly.

The other problem is how to incorporate ST code generation in the current industrial workflows. The PLC programming can be hardware-specific, and have I/O mappings and real-time requirements, which are to be maintained in any automated system. Therefore, the use of LLM-based code generation tools in this situation needs an in-depth knowledge of the language and the industrial setting. There are few studies who used LLM for industrial settings.

Sousa et al. [100] described the implementation of an IEC 61131-3 compiler in MatPLC, an open-source programmable logic controller platform. As per the IEC 61131-3 standard, computers that are a part of the industrial automation support languages such as Instruction List (IL) and Structured Text. The compiler, presented here, focused on Instruction List (IL) and Structured Text standards, which enabled users to implement standard compliant PLC programs in the MatPLC platform. The system is designed in a modular format to ensure that it can function without any hitches in different POSIX-compliant platforms. The system consisted of autonomous modules for managing logic execution, I/O, user interfaces, etc., exchanging data through shared PLC state variables. The modular approach helped to achieve interoperability and was easy to integrate different components, i.e., Python-enabled scripts or graphical interfaces, into the system. The method of logic of the compiler followed a usual multi-stage process including lexical analysis, parsing, semantic analysis, and code generation. Source code was converted to an abstract syntax tree first and compiled to C++ code using GCC. By using this design, the compiler has all the necessary tools to migrate into other platforms and could easily be integrated into GCC later. When working on the compiler, the development team encountered several ambiguities and inconsistencies of the IEC 61131-3 specification. Some of these issues include namespaces definition issues, conflicting operator definitions, lack of array support in function blocks, and confusing rules about task initialization semantics. The authors suggested changing the standard to correct these ambiguities and to make the language clearer. The compiler is now working, although it only supports core language features, there is still work to be done with the semantic checker and full config mapping. This

work helped promote open automation by allowing the practical implementation of standard compliant control programs on non-proprietary platforms. Future work includes completing the semantic checker, extending graphical language capabilities, and integrating more with the MatPLC runtime.

Staroletov et al. [101] presented the design and the assessment of the proposed process-oriented Structured Text (poST) as an extension of the IEC 61131-3 ST language for process control in industrial automation. The present work aimed at evaluating the correctness of the poST-to-ST translation by employing grammar-based fuzz testing. The authors used Xtext framework to develop tools for parsing and converting poST code to ST and then tested it using SynTESK tool for generating positive as well as negative test cases. The findings showed that grammar-based testing could be used to detect syntax and transformation errors that other forms of unit testing cannot. The paper also discussed the further improvements such as the automation of the grammar conversion and inclusion of a more comprehensive ST parser into the CI/CD process.

Koziolek et al. [60] analyzed the possibilities of using ChatGPT (GPT-4) as a source of control logic for industrial automation, especially for PLCs and DCS using IEC 61131-3 Structured Text. The authors developed and validated one hundred prompts in areas of basic control engineering problems, including standard algorithms, process control, interlocks, and diagnostics. The research also revealed that ChatGPT can produce syntactically correct and often concise ST code and demonstrated understanding of industrial automation and communications. However, some of the disadvantages include occasional mistakes, lack of ability to write in graphical notation such as Ladder Logic, and token limitations for larger programs. The authors argued that LLMs could be beneficial to control engineering in that they could help in code generation, debugging, and documentation yet must be validated. They also suggested more research on benchmarking LLMs for control logic generation, the enhancement of the prompt engineering process, and the analysis of LLM's effects on the engineering processes.

Yang et al. [121] presented AutoPLC, an LLM-based framework for generating ST code for PLCs and dealing with issues in vendor-specific ST and its verification. Since different vendors such as Siemens, Allen-Bradley, and CODESYS use customized versions of ST, AutoPLC was equipped with a Retrieval-Augmented Generation (RAG) module and an adaptive code checker in order to generate precise ST code compliant with the chosen vendor. Rq2ST knowledge base and Instruction library helped the LLM to find appropriate examples and ensured that code produced was syntactically and semantically correct. An improvement loop applied based on the given feedback from the code checker. When compared AutoPLC with seven bench-

marks, including open source ST and Siemens SCL variants, it offered higher correctness and compiled performance. These results showed that knowledge retrieval and iterative error correction were key to enhance the ST code generation. Further development of the tool will include integration with other PLC programming languages as well as fine-tuning of the verification process in order to increase code dependability.

Liu et al. [70] proposed Agents4PLC, an LLM-based multi-agent system that aimed at the automatic generation and validation of PLC code for ICSs. Unlike the previous models that based on design level checking, Agents4PLC checked code level correctness using a closed loop of Retrieval, Planning, Coding, Debugging, and Validation agents. Combining the use of RAG, prompt engineering, and Chain-of-Thought reasoning that enhanced the effectiveness of the generated code. Comparing the results of tests with the benchmark dataset, it could be stated that Agents4PLC was more efficient in comparison with such previous tools as LLM4PLC, as it provided better syntax validation, higher functional correctness, and more automation. It was implemented in real-world industrial scenarios, and the results showed that the system was able to produce correct and executable PLC code.

Tran et al. [109] focued on the usage of Large Language Models in the generation of PLC code in ST, which was one of the IEC 61131-3 defined programming languages. The authors compared general-purpose LLMs including ChatGPT-3.5, ChatGPT-4, Bard, Code Llama, StableCode, and Platypus2 in their correctness and capability to generate correct, executable SCADA code for 21 real-life control logic scenarios. They used CodeBERT Score to measure the syntactic similarity and pass@k for a functional correctness check, and all the generated outputs crosschecked using CODESYS for executability. They introduced a two-stage evaluation e.g, before and after prompt optimization. The prompt engineering enhanced the performance of all the models in the study. Based on the findings, ChatGPT-4 is deemed the best model as it performed highest execution accuracy when using the optimized prompts. ChatGPT-3.5 outperformed the others, while Bard though syntactically similar to it often provided non-executable code. Some open-source models such as Code Llama and Platypus2 faced challenges in terms of syntax limited support. The study showed that there was a high level of performance of general-purpose LLMs for generating PLC code but that the prompts needed to be designed carefully and the resulting code validated manually. However, the following limitations are worth noting in the study: the prompt dataset is quite small, and it includes only 21 cases; the evaluation was carried out with regard to only the ST language; local models used in the experiments were quantized due to the limitations of the hardware. The program correctness checked manually, without the use of specific tools

in the CODESYS environment.

Han et al. [48] developed a domain-specific translation tool for ST into equivalent readable C code. Since the prior work of translating ST-to-C have issues like the poor quality of the generated code, the authors used LLMs for post-processing the C codes to make the generated codes more maintainable through mnemonic naming, variable mapping, and inline documentation. The tool aligned with the standard compiler design phases, including lexical analysis, syntactical analysis, semantic analysis, and code generation. The real-world examples made it highly relevant to industrial use. However, the paper has certain limitations; it may fail for complicated or vendor-specific ST constructs, its applicability to other non-standard dialects is rather restricted, and the usage of LLMs for code enhancement may cause non-deterministic results in terms of the output quality. However, the scalability to large-scale industrial systems still remained an open question. Still, the presented approach offered a substantial contribution to the integration of PLC code with other systems in industrial automation.

Fakih et al. [36] introduced LLM4PLC, an iterative process based on the LLM that aimed at synthesizing and checking the PLC code for industrial control systems. While the existing LLM-based code generation did not ensure execution and formal verification. The LLM4PLC incorporated user feedback and external checkers, compilers, and SMV for improved correctness. The proposed framework consisted of the use of prompt engineering and LoRA fine-tuning for enhancing the code synthesis. The experiments with GPT-3.5, GPT-4, and the Code Llama models revealed that LLM4PLC increased the code generation success rate from 47% to 72%, thus, increased code accuracy and quality according to human coders. It was tested employing a FischerTechnik Manufacturing Testbed (MFTB) to demonstrate the generation of verifiable and executable PLC code. Their study showed how LLM4PLC contributed to minimize the engineering effort, enhanced code reliability, and increased the speed of industrial automation. Subsequent studies should seek to fine-tune the verification processes and improve the generality of models for the purpose of application in different industries.

Haag et al. [46] proposed a new fine-tuning approach for Large Language Models to generate IEC 61131-3 ST code with the help of compiler feedback and preference learning. Their approach made RLHF inadequate given that the language style used by ST was complex and there were not many datasets on which to train the model. The proposed idea for integrating the two paradigms SFT and DPO to use synthetic experts and compiler feedback for enhancing syntactic and semantic quality. The experimental result indicated an increase in the successful compilation of programs from 7% to 70% and semantic correctness up to 45%. This approach showed potential

for developing the domain-specific AI tools for industrial automation.

Automated ST code generation is still at its beginning. Research is also complicated by the fact that there are no standard benchmarks and evaluation metrics. The majority of them lack fine-tuning on ST or testing on ST-specific tasks, which results in inadequate performance and low generalizability.

2.5 Identified Gaps and Challenges in Existing Research

The literature points to a few gaps and issues in the application of LLMs to code generation, especially in niche and resource constrained environments such as Structured Text programming in industrial applications.

- Computational Resource Constraints:

  State-of-the-art LLMs such as GPT-3 and GPT-4 require extensive computational infrastructure, often involving hundreds or thousands of GPUs for training and multiple GPUs for inference.

- Lack of Domain-Specific Models:

  Most LLMs are trained on general-purpose code or natural language corpora. As a result, they lack familiarity with specialized domains like ST programming.

- Limited Public Datasets for ST:

  The absence of large, open-source ST datasets hampers the development and evaluation of LLMs in this domain.

- Evaluation Challenges:

  Standard code generation benchmarks like HumanEval or MBPP do not include tasks related to ST or industrial automation.

Based on the targeted thematic review conducted in this chapter, several limitations and open challenges emerge that are particularly relevant for domain-specific code generation in industrial automation. Although current literature shows impressive advances in the creation and use of LLMs to generate code, several gaps and yet to be solved issues are clear. First, the existing works has focused on general-purpose programming languages (e.g., Python, Java, or C++), and domain-specific languages (such as Structured Text) have received little attention even though they have a great importance

in the industry. This asymmetry has restricted the use of state-of-the-art code LLMs in the automation and control engineering setting. Second, even though recent NL-to-code approaches have achieved high syntactic quality, semantic correctness, robustness, and generalizability remain a challenge to generate code across a variety of tasks and industrial settings. Further, the majority of the evaluations are based on benchmark datasets which do not reflect the full complexity of the real world and, as a result, limit the validity of the performance measurements. Another issue is the high computation requirements to train and deploy LLMs that limit their accessibility and adoption in limited resource industrial environments. These gaps motivate the methodological and empirical contributions presented in Chapter 3, Chapter 4, and Chapter 6 which focus on developing resource-efficient, privacy-preserving LLMs and creating a pipeline for synthetic ST code dataset in industrial settings.

Chapter 3

Industrial Feasibility of PEFTpyCoder: A Parameter-Efficient Fine-Tuned Model for In-House Code Assistance

This chapter is based on a peer-reviewed publication [90] and presents an experience report on an industrial feasibility study that evaluates the application of code LLM technologies within an organization, ensuring that no data leaves the firm's perimeter. To address a set of feasibility questions, this study employed state-of-the-art techniques, including Low-Rank Adaptation (LoRA), Parameter-Efficient Fine-Tuning (PEFT), and 4-bit quantization, to develop a pipeline for creating a fine-tuned version of an existing code LLM that is small enough to allow both training and inference on commodity hardware, thereby eliminating the need for cloud-based solutions that would expose sensitive assets, such as parts of the existing code base, outside the firm's local network. The pipeline evaluated on a general programming language (namely Python) to assess its performance, and results show that it is adequate for the task at hand. Another motivation for this study stems from the firm's adoption of a domain-specific programming language, which is not directly supported by most existing code LLMs. The lessons learned in this experience could be valuable in many other contexts where a firm is evaluating the feasibility of an in-house AI-based solution rather than depending on external service providers.

3.1 Introduction

Code LLMs [23, 112] are advanced AI models trained specifically on vast amounts of programming code. These models can understand, generate, and assist with code-related tasks such as code completion, bug fixing, and code translation across different programming languages.

Among the diverse industrial applications of generative AI, the integration of code LLM-based AI assistants has emerged as a powerful strategy for enhancing developer productivity, improving code quality, and accelerating software development cycles.

As with other similar LLM applications, organizations considering the adoption of these technologies face a critical decision, i.e., whether to rely on a specialized service provider, or to fine-tune an existing code LLM in-house. Each approach offers distinct advantages and challenges that must be evaluated based on the organization's specific needs, resources, and constraints.

One of the primary advantages of utilizing an existing AI-based coding assistant from a service provider is the ease of implementation. These solutions are typically pre-trained on large datasets encompassing multiple programming languages and development environments, enabling them to deliver robust and sophisticated functionality out of the box.

By adopting a third-party service, organizations can bypass the substantial time and resource investments required for model training, fine-tuning, and ongoing maintenance. Service providers also handle infrastructure requirements, ensuring that the AI system scales effectively with the organization's needs. The adoption of third-party services is particularly attractive for smaller companies, or those with limited AI expertise, as it allows for quick deployment and immediate benefits.

However, relying on an external service provider comes with several considerations. Data security and privacy are significant concerns [32], as these services often require access to proprietary code bases or other sensitive assets. While providers like OpenAI and Google implement strict security protocols, the need to transmit data outside the organization's controlled environment can still pose risks, especially for firms in highly regulated industries or dealing with intellectual property that demands stringent confidentiality.

Furthermore, fine-tuning capabilities with these services are often limited. While in-house fine-tuning allows for deep customization, existing service providers typically offer fine-tuning options in more restricted forms, such as RAG [65]. RAG leverages an external knowledge base to tailor the AI's output to specific queries but does not fully customize the underlying model to the organization's unique codebase. This limitation may result in a less optimized performance for specialized coding needs, as the model's suggestions

are based on more general-purpose datasets.

In contrast, fine-tuning an existing code LLM within the organization's own infrastructure offers a higher degree of customization and control. This approach allows the AI model to be better suited to the organization's coding standards, languages, and development practices, potentially leading to more accurate and relevant suggestions. Fine-tuning involves further training the model on proprietary code, ensuring that the AI assistant is well-versed in the organization's specific codebase and development environment.

However, this in-house approach is not without challenges. The process of fine-tuning a code LLM is computationally intensive and requires significant expertise in machine learning and natural language processing. Organizations must invest in the necessary hardware, such as GPUs or TPUs, and allocate resources for the ongoing maintenance and updating of the model to keep it aligned with evolving coding practices and security standards. Despite these investments, the resulting AI assistant can be finely tuned to the organization's specific needs, offering a level of precision and relevance that external services may not achieve.

While in-house fine-tuning mitigates some of the data privacy concerns associated with third-party services, it does not eliminate them entirely. Ensuring that the training process is secure, and that sensitive data remains protected, requires rigorous internal protocols and even regulatory compliance, depending on the industry.

The present study arises from the need to address these concerns within the context of Coesia, a leading industrial firm operating worldwide in the packaging and automation market. Software plays a relevant role in the machines marketed by the firm, with a significant part of it being written in Structured Text (ST) [1], a domain-specific language used to program the complex automation provided by their machines. Coesia is also extremely strict in its policies regarding the access of its assets, which include the source code for their software. The presence of the need for a code LLM able to "understand" a language for which existing general-purpose code LLM have no support for, and the cautious approach to private assets, make the in-house solution particularly attractive.

To assess its viability, we structured our study around the following Feasibility Questions.

FQ1. Can an existing code LLM model be fine-tuned in-house on commodity hardware while achieving state-of-the art performances?

[1] IEC 61131-3 describes software architecture and acceptable programming languages to use within a PLC. The only non-deprecated textual programming language currently is Structured Text

FQ2. Can the model be deployed locally or as an intra-organization service to implement an AI coding assistant?

FQ3. Are the licenses under which existing code LLM are released compatible with the organization's constraints?

FQ1 addresses on the in-house training of fine-tuned models. FQ2 focuses on the delivery of the AI-based solution via *inference*, the application phase of LLMs where their learned capabilities are used to interpret and respond to inputs. Deployment can take the form of a software component that runs locally on users' workstations or a software service accessible within the organization via local components. FQ3 highlights a critical, yet often overlooked, concern when deploying such solutions. Many state-of-the-art models are publicly available and can easily be downloaded and used as a base model for fine-tuning. However, they often present licenses that limit the applicability of the base model and its fine-tuned versions, typically regarding commercial use, modifications, redistribution, and applicability to specific domains or ethically sensitive contexts.

While these questions arise from the interests of a specific company, we believe they address topics relevant to a wide range of organizations, including but not limited to those operating in industrial sectors.

The remainder of the paper is organized as follows: Section 3.2 analyzes the potential and limitations of AI in manufacturing. Section 3.3 explores commercially available GenAI-based code assistants. Section 3.4 reviews some recent literature related to LLM-based code generation. Section 3.5 discusses licensing issues related to the distribution and use of LLMs, which are very important in an industrial setting. The proposed methodology is presented in Section 3.6, and its performance is evaluated in Section 3.7. Section 3.8 presents a discussion of the results. Section 3.9 highlights the threats and potential limitations of our study. Finally, Section 3.10 draws some conclusions and highlight possible future research directions.

3.2 Generative AI in Manufacturing

Generative AI has the potential to revolutionize the manufacturing industry by enabling people to obtain better results, faster. While results are still under evaluation, the potential benefits are clear and as such the industry is investing heavily in this technology. We present here a brief survey of the current state of the art in Generative AI adoption in the manufacturing industry.

Urlana et al. [110] discussed the possibilities and challenges in LLMs deployment depending on the scale of the firms. Small-scale firms rely on prompt engineering and transfer learning while large firms on continual pre-training and collaborative tools. They also explained why 'one-size-fits-all' approach for adoption of LLMs into industry does not work: small, medium and large organizations face different requirements and challenges concerning resources, guaranteed rules, regulations, and technical conditions of the working environment, data protection, ethical aspects, and infrastructure.

Fui-Hoon Nah et al. [78] explored the broad usage of generative AI in industrial sectors like business, education, healthcare, and content business, including ethical, technical, and regulatory problems. They recommended establishing a symbiosis between man and AI, called human-centered artificial intelligence and human-in-the-loop systems, as the resolutions of AI ethical issues.

Singh et al. [99] examined how Generative Artificial Intelligence influenced the performance of organizations aiding exploratory (new ideas) and exploitative (improving existing processes) innovation. They also discussed how ethical issues, and environmental dynamism moderate these impacts. They concluded that GenAI improved both forms of innovation mainly in dynamic environments and advised organizations to introduce conducive ethical principles to govern AI usage for productivity.

Kar et al. [59] presented the prospects, opportunities, and limitations of GenAI in industries. The innovative applications of the technology, such as GPT-3/4 and BERT, might be embedded into organizations to aid decision-making, process automation, as well as improving customers' experiences.

3.3 Commercial GenAI-based Coding Assistants

The advent of generative AI has triggered the development of sophisticated coding assistants, that leverage LLMs to augment software development processes. This section provides a technical overview of several prominent AI-based coding assistants currently available in the market, independently of their applicability to our specific use case.

- **ChatGPT** [82], developed by OpenAI, is an advanced AI language model designed to assist with a variety of tasks. Leveraging the GPT-4 architecture, ChatGPT supports multiple programming languages and can generate code snippets, provide explanations for existing code, and assist in debugging by identifying potential issues and suggesting fixes.

Its versatility makes it a valuable tool for developers seeking assistance across different stages of the software development lifecycle. If asked to write Structured Text code, it can write some snippets, but usually the code quality is very low and rarely passes the compilation step. It is also not integrated into any development environment.

- **GitHub Copilot** [39], a product of GitHub, serves as an AI-driven code completion tool. It assists developers by predicting and suggesting code snippets and entire functions in real-time. Launched in 2021 using the Codex model from OpenAI, Copilot integrates with many development environments (IDEs). It is trained on an extensive dataset comprising public code repositories, is accessible via a subscription model, and supports the most common programming languages. Recent versions have added support for diversified LLMs, such as Anthropic's Claude or OpenAI o1, in order to overcome the limitations of the previous LLM.

- **Devin AI** [28], developed by Cognition Labs, is an autonomous AI assistant designed to perform software engineering tasks such as coding, debugging, planning, and problem-solving. Users can prompt Devin AI with tasks in natural language, and the tool responds by generating plans and implementing code, adjusting its approach based on user feedback. Devin AI represents a significant advancement in AI-driven software development, aiming to handle complex coding tasks with minimal human intervention. More than a focus on the code itself, the tool excels in full development-cycle activities.

- **Google Gemini** [41] is an AI coding assistant capable of coding in more than 20 programming languages, including Python, Java, C++, and JavaScript. It can generate code from prompts and comments, explain code snippets, and assist in refactoring existing code. Gemini (previously known as Bard) leverages Google's expertise in AI to provide developers with a tool that enhances coding efficiency and understanding.

- **Anthropic's Claude** [9] is an AI assistant that incorporates "Constitutional AI" to set safety guidelines for its output. The latest models, including Opus, Sonnet, and Haiku, offer capabilities such as coding assistance, text generation, and image input processing. Claude is integrated into platforms like Amazon's Bedrock, providing developers with advanced AI-driven coding support.

- **TwinCAT Chat** [13], developed by Beckhoff Automation, is trying to integrate LLMs into the TwinCAT engineering environment. This integration is aimed towards facilitating AI-assisted engineering tasks, including automated code generation, optimization, documentation, and refactoring. Accessible via a chat window within Visual Studio, TwinCAT Chat connects to the cloud host of the selected LLM, enabling seamless communication between the user and the AI model. Future developments aim to expand its capabilities, such as automating the creation of Human-Machine Interface (HMI) controls and configurations, further simplifying the engineering process. As of today, it is still quite limited by the capabilities of the underlying model, without taking into consideration any kind of offline usage that is not possible.

Despite the widespread availability of these kinds of tools, here presented in a most-known subset, none of them today is able to help PLC programmers write high-quality code in Structured Text language, nor is it available as an offline customizable environment.

3.4 LLM-based Code Generation

The literature review for this study is already summarized in Chapter 2 Section 2.3. The original paper [90] included its own discussion of prior work, which has been condensed here to avoid redundancy.

3.5 Licensing considerations

While LLMs from major companies dominated in the recent past, smaller, specific, vertical, application-oriented Small Language Model (SLM)s are now being released at a rapid pace, with many new entities increasingly contributing to the field.

This quickly evolving landscape has led to some confusion, especially around the differences between open-weight and open-source models. In fact, closed-source models are probably the easiest to understand. Their development is usually controlled by a single entity or company. They are often available solely via an API, and cannot be modified or redistributed without permission. Their Licenses can be restrictive, and they may not be suitable for all use cases, e.g., it is typically not permissible to legally utilize data generated by closed-source models to customize a separate model in order to reduce costs.

Open-weight models make their neural network weights public but often have restrictive licenses. These licenses can limit modification, distribution, and commercial use. In contrast, open-source models are freely available to use, modify, and distribute, including for commercial purposes, provided that compliance with the license is maintained.

Beyond standard practices in software engineering, the adoption of AI-driven tools in corporate environments opens up new challenges related to the content produced by the tool. The use of Generative AI raises questions about IP ownership in derivative works: permissive licenses do not claim ownership over them, reducing legal risks.

Compliance with the licenses of the models and datasets used in the development of a solution is crucial for its adoption in a corporate environment. Given the rapid pace of change, it is essential to continuously verify the conditions under which any downloadable resources are provided. The enormous abundance of data and models shrinks when applying these rigid filters. If this passage is skipped, the risks would be double faced: on one side, a tool or knowledge that is not permitted for use in the given context may be incorporated into the pipeline; on the other side, it may become necessary to release the work under the same license as the part adopted in the solution. This might have serious consequences, both on the competitive advantage being pursued, which may become available to everyone, and to the fact that the published model might leak the sensitive material it was trained on before understanding the implications and the requirements of republishing it.

In order to be adopted in an enterprise environment, the solution had to be compliant with the requirements described above. These include commercial compatibility and intellectual property ownership. For this experiment, we restricted ourselves to the use of BSD, MIT or Apache licensed models and datasets, since these licenses are permissive and allow reuse within proprietary software. This choice was made to ensure that the solution could be easily integrated into corporate infrastructure and to avoid potential legal issues.

Table 3.5.1 shows model families that we found suitable for our purposes.

Table 3.5.1: Licenses of open LLMs

Language Model	**License**
T5[2]	Apache2.0
Pythia[3]	Apache2.0
Dolly[4]	MIT
OpenLLaMA[5]	Apache2.0
Falcon[6]	Apache2.0
GPT-J-6B[7]	Apache2.0
Mistral7B[8]	Apache2.0
Grok-1[9]	Apache2.0
Mamba-7B[10]	Apache2.0
Mixtral8x22B[11]	Apache2.0
Phi-3[12]	MIT

Also some code-dedicated models are available under the same license set (Table 3.5.2); note that both have been created by Salesforce.

Table 3.5.2: Licenses of open LLMs for code generation

Language Model	**License**
CodeT5+[13]	BSD-3-Clause
CodeGen2.5[14]	Apache2.0

Finally, a few datasets for training with somewhat clear license compliant with the requirements described above are listed in Table 3.5.3.

[2] https://github.com/google-research/t5x/blob/main/docs/models.md
[3] https://github.com/EleutherAI/pythia
[4] https://huggingface.co/databricks/dolly-v2-12b
[5] https://huggingface.co/openlm-research/open_llama_3b
[6] https://huggingface.co/tiiuae/falcon-7b
[7] https://github.com/kingoflolz/mesh-transformer-jax/
[8] https://huggingface.co/mistralai/Mistral-7B-Instruct-v0.1
[9] https://huggingface.co/xai-org/grok-1
[10] https://huggingface.co/TRI-ML/mamba-7b-rw
[11] https://huggingface.co/mistralai/Mixtral-8x22B-Instruct-v0.1
[12] https://huggingface.co/microsoft/Phi-3-mini-128k-instruct
[13] https://github.com/salesforce/CodeT5/tree/main/CodeT5+
[14] https://huggingface.co/Salesforce/codegen25-7b-multi
[16] https://www.together.xyz/blog/redpajama
[17] https://huggingface.co/datasets/codeparrot/apps

Table 3.5.3: Licenses of open datasets for LLM training or finetuning

Name	**License**
RedPajama[15]	MIT, BSD, or Apache licenses only[16]
APPS[17]	MIT

3.6 Methodology

We now discuss the methodology used to address the feasibility questions from Section 3.1. We chose to fine-tune an existing LLM to enhance its performance in generating code from natural language prompts. Our goal is to evaluate our ability to achieve state-of-the-art results using limited resources that a generic company might already have in-house. It is important to note that our fine-tuning focuses on the Python programming language rather than on Structured Text, which is the language of interest to Coesia. The reason for this is that we present a feasibility experiment: our aim is to assess the viability of in-house training, with the intention of applying the same approach to develop a tool for internal deployment if the preliminary results prove promising. The construction of a high-quality training dataset is likely to be, in fact, the most costly aspect of fine-tuning, and we want to pursue this effort only after validating the approach. Additionally, while one goal is to assess whether training can be performed in-house, our experiments are conducted using Cloud resources. Although this may seem counterintuitive, it is important to note that we do not have confidentiality restrictions on the training dataset, since it is already publicly available. We made sure that the virtual resources used have similar characteristics to general-purpose GPUs available in the consumer market.

3.6.1 Model Architecture

We adopted the pre-trained phi-3-mini-128k-instruct [1] model with 3.8 billion parameters, trained on 3.3 trillion tokens. It is a transformer decoder model with a context length of 128K and a tokenizer size of 32,064. The model has 3072 hidden dimensions, 32 heads, and 32 layers; the `bfloat16` data type is used for training. We applied PEFT to fine-tune our model on an instruct dataset for code generation from natural language prompts. We used LoRA, which is the type of PEFT fine tuning in our proposed model. Fig. 3.6.1 depicts the pipleine diagram of the experiment.

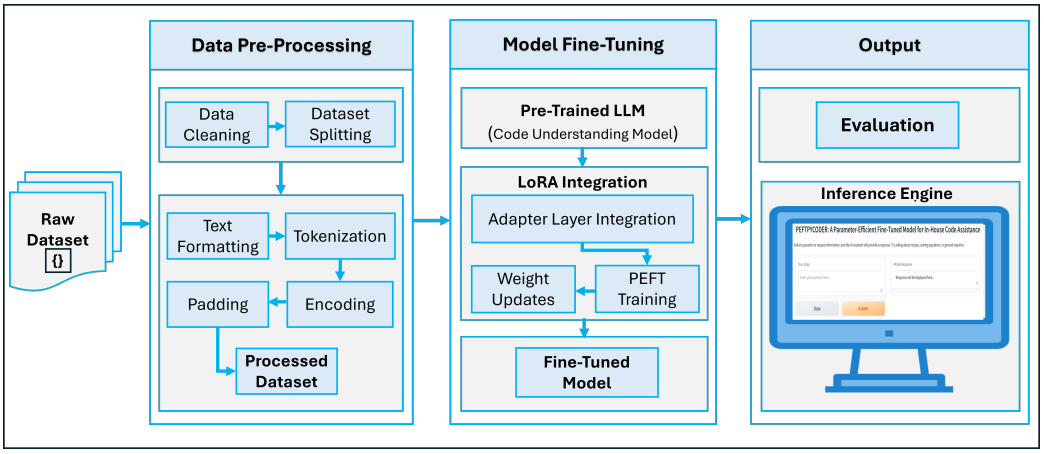


Figure 3.6.1: Pipeline of the Training Experiment.

3.6.2 Parameter-Efficient Fine-Tuning (PEFT)

PEFT [49] is a NLP method that uses task-related information to fine-tune only a few of the pre-trained language model parameters, to improve its performance for a particular task. It "freezes" most layers and only trains the last few layers, thus requiring lower computational resources and less time.

3.6.3 Low-Rank Adaptation (LoRA)

LoRA [52] is a PEFT fine-tuning technique for Large Language Models that involves inserting small trainable modules into the transformer architecture. LoRA freezes the pre-trained model weights and injects rank decomposition matrices, which makes LoRA occupy a relatively small number of trainable parameters and GPU memory. This method can help to reduce trainable parameters up to 10,000 times and GPU memory requirements by three times, while at the same time either improving the performance of the model on different tasks, or maintaining it at a level comparable to the original model's performance.

Fig. 3.6.2 provides details of how Low-Rank Adaptation layers operate within a transformer model. It shows two key components: a down-projection path, which lowers the dimensionality of the input for computation, and an up-projection path which brings back the features to its original dimension after some transformations. This mechanism allows the model to fine-tune effectively due to update isolation of needed layers only, hence reducing the utilization of computational power, and help to achieve desirable performance. Fig. 3.6.3 gives an insight into the dimensionality transformation

of the Low-Rank Adaptation framework.

For completeness, we provide below a mathematical background on the LoRA process.

Given an input $\mathbf{x} \in \mathbb{R}^{d_{\text{model}}}$, let:

$$\mathbf{h} = \mathbf{W}_2\sigma(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2 \tag{3.1}$$

where $\mathbf{W}_1 \in \mathbb{R}^{d_{\text{FFW}} \times d_{\text{model}}}$, $\mathbf{W}_2 \in \mathbb{R}^{d_{\text{model}} \times d_{\text{FFW}}}$, $\mathbf{b}_1 \in \mathbb{R}^{d_{\text{FFW}}}$, $\mathbf{b}_2 \in \mathbb{R}^{d_{\text{model}}}$, and σ is a non-linear activation function.

$\mathbf{A}$ and $\mathbf{B}$ are two low-ranked matrices introduced by LoRA, such that $\mathbf{A} \in \mathbb{R}^{r \times d_{\text{model}}}$ and $\mathbf{B} \in \mathbb{R}^{d_{\text{FFW}} \times r}$, where $r \ll d_{\text{model}}, d_{\text{FFW}}$. The feed-forward layer with LoRA adaptation is given by:

$$\Delta\mathbf{h} = \mathbf{B}\sigma(\mathbf{A}\mathbf{x}) \tag{3.2}$$

The final output $\mathbf{h}'$ of the feed-forward layer, incorporating the LoRA adaptation, is:

$$\mathbf{h}' = \mathbf{h} + \Delta\mathbf{h} \tag{3.3}$$

where $\mathbf{h}$ is the original output computed by the feed-forward layer, and $\Delta\mathbf{h}$ is the low-rank adaptation term.

By incorporating LoRA into the original feed-forward process, we have:

$$\mathbf{h} = \mathbf{W}_2\sigma(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2 \tag{3.4}$$
$$\Delta\mathbf{h} = \mathbf{B}\sigma(\mathbf{A}\mathbf{x}) \tag{3.5}$$
$$\mathbf{h}' = \mathbf{h} + \Delta\mathbf{h} \tag{3.6}$$

from which we derive:

$$\mathbf{h}' = \mathbf{W}_2\sigma(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2 + \mathbf{B}\sigma(\mathbf{A}\mathbf{x}) \tag{3.7}$$

By using these equations, Low-Rank Adaptation efficiently introduces a minimal number of additional parameters $\mathbf{A}$ and $\mathbf{B}$ to fine-tune the model for specific tasks while maintaining computational efficiency.

3.6.4 Dataset

We used APPS [51] dataset for fine-tuning our model. APPS is a code generation dataset with questions and answers containing a total of 10,000 problems to check language models' ability to generate code from the natural language descriptions. Each sample consists of a problem description in English, ground truth Python code, test cases, names of functions to solve (if given), levels of difficulty, and source information. The dataset includes

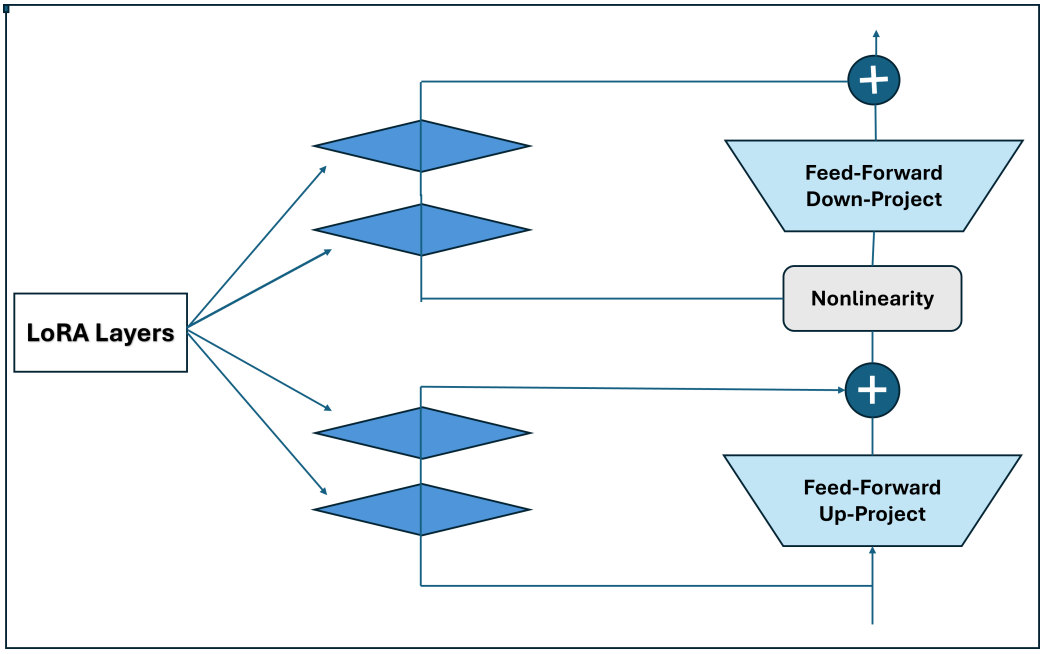


Figure 3.6.2: This diagram illustrates the architecture of a model incorporating LoRA (Low-Rank Adaptation).

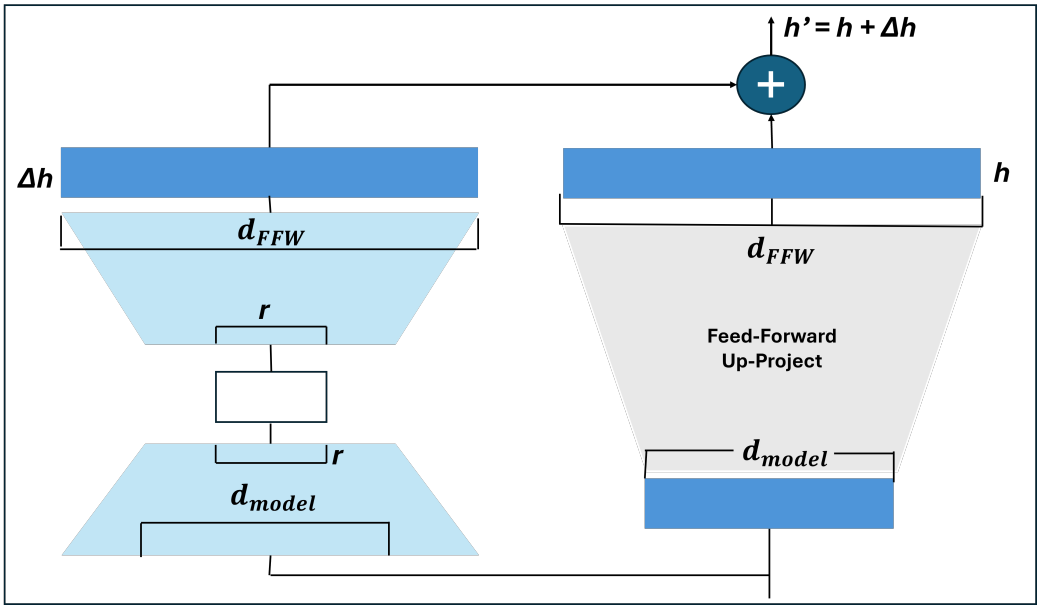


Figure 3.6.3: LoRA Finetuning.

problems from Codewars, AtCoder, Kattis, and Codeforces with 5000 samples for the training set. The APPS metric is located on the CodeParrot

hub. The average length of a problem is 293.2 words. Before feeding the data to the model for training, we pre-processed the dataset by creating a chat template by combining the question and answer fields of the dataset to feed the input to the model.

3.6.5 Training

We used Amazon Web Service (AWS) cloud infrastructure to fine-tune our model, in particular we used AWS g5.xlarge EC2 instances. We used the accelerate library by Hugging Face to accelerate the training and manage memory more effectively. The model was fine-tuned on a single NVIDIA A10G GPU with 24GB RAM. We used only 0.65% of the model's parameters during fine-tuning and we applied Low-Rank Adaptation configuration to all linear modules of the model. Finally, we used the following framework versions for our experimental setup:

- Pytorch (v. 2.3.0+cu121)
- HuggingFace Parameter-Efficient Fine-Tuning (v. 0.11.0)
- HuggingFace Transformers (v. 4.40.2)
- HuggingFace Datasets (v. 2.19.1)
- HuggingFace Tokenizers (v. 0.19.1)

The Low-Rank Adaptation configuration and hyperparameter settings used in our training are described in Table 3.6.1 and Table 3.6.2 respectively.

A functional description of the structure and data flow of the proposed model is shown in Fig. 3.6.4. This diagram illustrates the fine-tuning process of a pre-trained model for downstream tasks of code generation. The pre-trained LLM further fined tuned using code dataset containing pairs of problem prompts and corresponding solutions. During the fine-tuning process, the model learns to solve the coding problems more efficiently and accurately, as shown in Fig. 3.6.4.

3.7 Model evaluation

We evaluated the performance of our proposed model through widely used benchmarks, e.g., HumanEval [23] and MBPP [11], which consist of hundreds of Python programming problems that use test cases to validate the code produced by a Code LLM. We applied the Pass@k metric (where k=1)

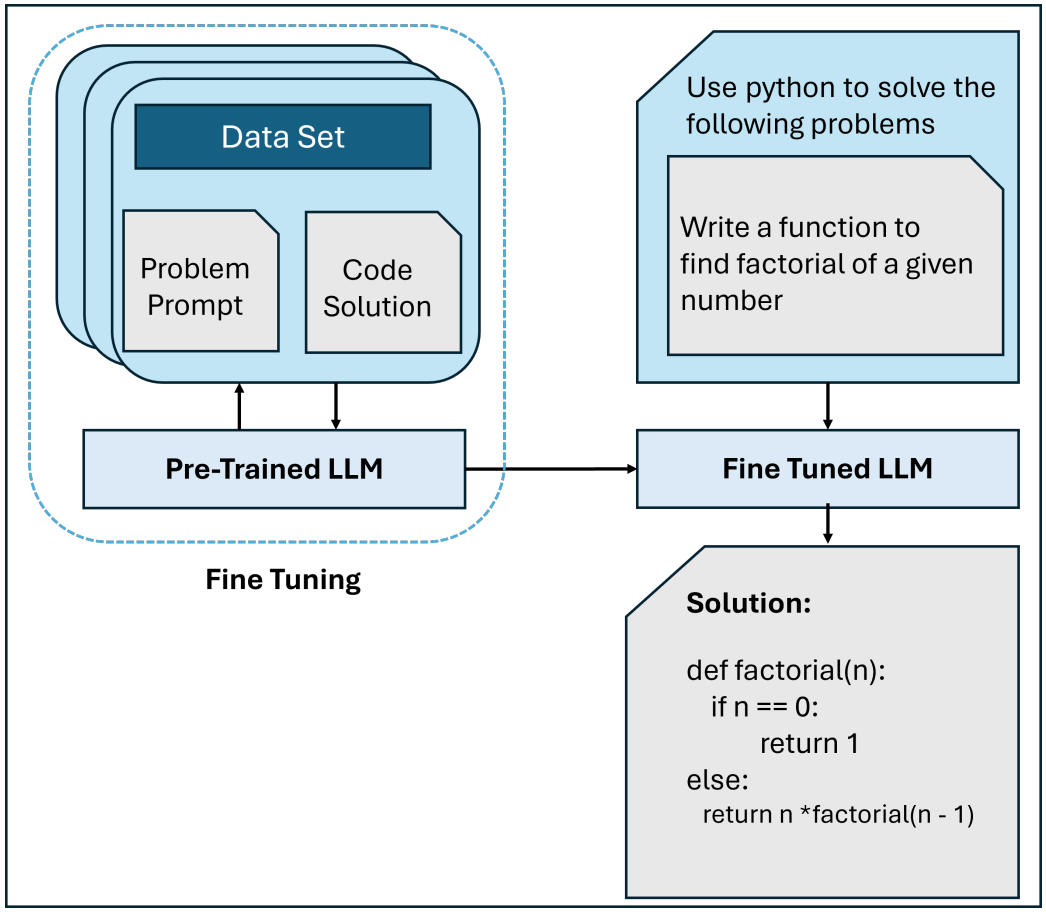


Figure 3.6.4: Model structure.

Table 3.6.1: LoRA Configuration.

Parameter	**Value**
Low-Rank Adaptation Alpha	16
r	32
Bias	None
Low-Rank Adaptation Dropout	0.05
Task type	CAUSAL-LM
Target Modules	qkv_proj, o_proj, gate_up_proj, down_proj

for evaluating the performance of Code LLMs with the same settings used in [23]. We used the BigCode Evaluation Harness [15] framework. This is a single-turn evaluation, and nucleus sampling is used with top-p being set to 0.95. We used the Pass@1 metric for evaluating the compared models

Table 3.6.2: Hyperparameters Configurations.

Parameter	**Value**
Warmup Steps	10
Per device train batch size	1
Per device evaluation batch size	1
Gradient Checkpointing	True
Gradient Accumulation Steps	16
Maximum Steps	100
Number of Epochs	1
Learning Rate	5.0e-06
Logging Steps	10
bf16	True
Optimizer	Adam [betas=(0.9, 0.999), epsilon=1e-08]
Seed	42
Learning Scheduler Type	cosine
Save Strategy	Steps
Save Steps	10
Evaluation Strategy	Steps
Evaluation Steps	10
Do Evaluation	True
Weight Decay	0.05

Table 3.7.1: Evaluation Settings for Benchmark Dataset.

Tasks	**HumanEval**	**MBPP**
Number of problems	164	500
Max Length Generation	1024	1024
Temperature	0.2	0.1
Load in 8 bit	True	True
Batch Size	10	10
Allow Code Execution	True	True

and generated one solution for each NL prompt. Table 3.7.1 illustrates the settings used to evaluate the performance of the state-of-the-art models.

We compared the performance of our proposed model with state-of-the-art, open-access code generation large language models. The results are reported in Table 3.7.2, and for better visual understanding also in Fig. 3.7.1. The values for other models are taken from [93] and [66]. We compared our

Table 3.7.2: Results on HumanEval and MBPP Benchmarks.

Model Name	**Size**	**HumanEval**	**MBPP**
SantaCoder	1.1B	18.0	35.0
CodeGen-Multi	16B	18.3	20.9
CodeGeeX	13B	22.9	24.4
CodeGen-Mono	16B	29.3	35.3
StarCoderBase	15.5B	30.4	49.0
StarCoder	15.5B	33.6	52.7
Code Llama	7B	33.5	41.4
Code Llama	13B	36.0	47.0
Code Llama - Python	7B	38.4	47.6
Code Llama - Python	13B	43.3	49.0
Phi-3-mini-128k-instruct	3.8B	60.4	52.5
PEFTPyCoder (our model)	3.8B	62.8	53.75

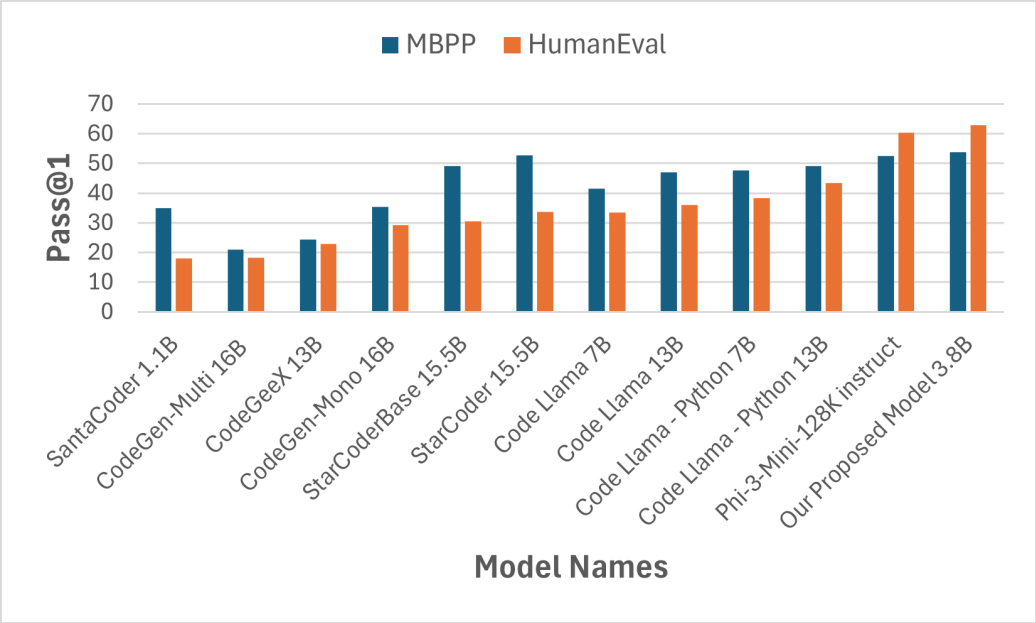


Figure 3.7.1: Results on HumanEval and MBPP Benchmark.

proposed model with alternatives up to $\sim 5\times$ larger, ranging from 1.1B to 16B parameters. As can be observed, our model compares favourably with the state-of-the-art, since it achieves superior performance using the lower computing resources on both benchmarks.

We also evaluated the raw performance of our model by measuring the

Table 3.7.3: Inference Performance

Model	Time to First Token (s)	Tokens/s
PEFTPyCoder-full	9.91	16.56
PEFTPyCoder-8bit	9.81	16.71
PEFTPyCoder-4bit	9.77	16.78

time to the first token generation and the total number of token generations per second for precision-scaled inference (e.g., full, 8-bit, and 4-bit). The results are shown in Table 3.7.3: our model is able to generate up to 16.78 tokens per second on a single NVIDIA A10G GPU with 24 GB of RAM.

3.8 Discussion

In this section, we answer the feasibility questions introduced in Section 3.1. Specifically, the evaluation presented in the previous section allows us to respond positively to question FQ1. Although by a small margin, we were able to fine-tune an LLM to improve its performance on a specific programming language.

A relevant consideration here is that the base model we adopted (as with most existing code LLMs) already exhibited strong performance on Python code, given the language's widespread use. We expect that fine-tuning for Structured Text will yield more significant improvements, as it is a much more domain-specific language (and thus less represented in the training data), while maintaining a syntactic structure similar to well-known languages like Pascal and Ada; this similarity should simplify its "structural comprehension" by the model.

Our training was conducted using a single NVIDIA A10G GPU with 24 GB of RAM, a reasonable hardware platform that most firms should be able to acquire without difficulty. Indeed, many commercial GPUs aimed at CAD and gaming domains offer comparable technical specifications.

Our second question, FQ2, addresses the deployability of an LLM-based solution, whether as a service or locally on developers' workstations. The relevant factors are the hardware required for deploying the inference module and the speed at which the model generates output. The data in Table 3.7.3 indicate that the model's inference speed is adequate for a single user when using the same hardware employed for fine-tuning.

However, in the case of deployment as a service, with multiple users accessing the model concurrently, the current solution would require a more powerful hardware platform. Conversely, local deployment, where the model

is instantiated on each developer's workstation, is entirely feasible. This is especially true in our case, as Coesia's developers' workstations and laptops are equipped with discrete GPUs that can accommodate the model (considering that the memory requirements for inference are significantly lower than for fine-tuning). Furthermore, even more resource-constrained platforms can be used with the aid of tools like llama.cpp [18], which, in our preliminary tests, demonstrated the ability to run Phi-3-based models with reasonable response times.

We believe this issue will be largely mitigated in the coming years with the widespread adoption of computing platforms that include Tensor Processing Units (TPUs) or Neural Processing Units (NPUs).

Our third question, FQ3, concerns licensing considerations. As discussed in Section 3.5, the adoption of AI-driven solutions in corporate environments raises new challenges related to content ownership and licensing. The models with permissive licenses we used in our experiments show that there are options available for corporate adoption, allowing us to respond positively to the feasibility question. However, we acknowledge that the legal landscape is complex and constantly evolving, and we recommend that firms consult with legal experts to ensure compliance with the licenses of the models and datasets.

3.9 Threats and limitations

As with many empirical studies, our experiments are subject to a number of limitations, both internal and external. Internal limitations pertain to common constraints associated with these kinds of experiments: limited training/validation dataset size and quality, limited evaluation metrics, and lack of human feedback. Although we employed some of the highest-quality datasets and state-of-the-art practices (Sections 3.6 and 3.7), the actual "usefulness" of an LLM-based solution is difficult to quantify through metrics alone. This limitation is shared by many other experiments; we hope that more refined, human-centered validation mechanisms will be developed to better assess the effectiveness of AI-based solutions.

Training set contamination is a potential concern. We argue, however, that the risk in our case is likely limited. The APPS dataset, used for fine-tuning, focuses on competitive programming problems, which differ significantly in scope and style from the benchmarks (HumanEval and MBPP) used for evaluation. Moreover, both benchmarks include tasks that require creative problem-solving and domain knowledge, reducing the likelihood that

[18] `https://github.com/ggerganov/llama.cpp`

performance is solely attributable to memorization. This risk could be further reduced with more rigorous data curation and filtering during pre-training and fine-tuning, which could help mitigate overlap.

Regarding external threats, the primary concern is generalizability. As previously discussed, one issue is that we assume the results achieved for a specific programming language can be replicated (and ideally improved) in another language. While we presented speculative arguments to support this belief, only actual fine-tuning on an ST dataset can provide a definitive answer. Nevertheless, we believe our approach is reasonable. One of the key motivations for this feasibility study is that building a high-quality training dataset is likely the most expensive aspect of fine-tuning, and we wanted to embark on such an endeavor only after validating our approach.

Additionally, we tested our fine-tuned model on specific question/answer tasks. While we expect a code assistant to provide other features such as code completion hints, the informal experiments conducted by us and others suggest that a fine-tuning mechanism effective for prompting typically works for other uses as well. However, we did not formally prove this in our experiments.

A further point worth considering is that our study focuses exclusively on developing an internal code assistance solution for the Coesia Company. This training approach may result in a model that is highly specialized in the coding styles, conventions, and problem-solving patterns specific to Coesia. While this specialization can be seen as a strength, as the model is tailored to the company's specific needs, it could also be a limitation. This narrow focus may restrict the model's utility and performance in broader applications.

3.10 Conclusion

In this feasibility study, we demonstrated the potential to develop an in-house AI-based code generation solution for the Coesia company by analyzing strategic and technical factors. We constructed a set of feasibility questions addressing the core phases like training, inference, and deployment to ensure that the solution is technically feasible, deployable, cost-effective, and compliant with licensing terms. Advancements in open-source large language models and fine-tuning techniques like Parameter-Efficient Fine-Tuning, Low Ranked Adaptation, and quantization techniques, it is possible to develop a custom AI-based solution using low computational resources allowing to incorporate Coesia's resources for data privacy control, customization, and compatibility with existing workflows. To validate our feasibility study, we also proposed PEFTPyCoder model, using Low-Rank Adaptation, for

Python programming. We observed that by applying Parameter-Efficient Fine-Tuning to the pre-trained model, we can get satisfactory results with a significant reduction of computational resources. We fine-tuned a pre-trained 3.8B parameter model for Python programming on the APPS dataset and compared the performance with state-of-the-art LLM-based code generation models ranging from 1.1B to 16B parameters. We found that our model achieved superior performance, suggesting that our proposed approach can be effectively applied in an industrial context and therefore deserves further investigation.

Further research will focus on adopting the same approach to fine-tune the model for the Structured Text language, while exploring the possibilities to expand the available dataset using synthetic data validated in the context of Coesia's codebase.

Chapter 4

Pipeline for Creating a Synthetic ST Dataset

The scarcity of high-quality, publicly available datasets in industrial automation languages such as Structured Text (ST) poses a significant challenge for the development and fine-tuning of Large Language Models (LLMs) tailored to this domain. Proprietary constraints, safety-critical requirements, and limited open-source code make it difficult to collect real-world ST programs at scale. To address this gap, this chapter presents a comprehensive pipeline for generating a large-scale *synthetic* dataset of ST programs, designed to support LLM training and evaluation. The pipeline integrates prompt engineering, LLM-based code generation, build validation using a commercial PLC compiler, mutation testing, and iterative refinement through automated feedback loops. This chapter describes the design goals, implementation details, and evaluation metrics used to ensure syntactic correctness, semantic richness, and industrial realism of the generated code. Empirical results show that the approach can produce thousands of diverse, compilable ST programs with an overall compile success rate above 99%, covering a wide range of control logic patterns. This work lays the foundation for scalable, domain-specific LLMs in industrial automation and offers a reproducible framework for similar efforts in other niche programming languages.

4.1 Introduction

Automation in industries revolutionized the modern manufacturing and production environment by providing exact, economical, and scalable control on physical procedures. PLCs are the heart of most industrial automation systems and are specialized computing devices that are used to run control

programs in a very reliable and real-time responsive manner [18]. Because of its strength, deterministic nature, and capability to communicate with industrial data communication systems, PLCs have become popular in many industries including automotive [47], energy [84], chemical processing [16], food production [106], manufacturing, and packaging solutions [75].

Another important aspect of PLC-based control systems is the Structured Text which is one of the five standard PLC programming languages defined by IEC 61131-3. Industrial automation relies heavily on domain-specific languages such as ST, defined by the IEC 61131-3 standard [31]. ST is a high-level programming language meant to write in a structured and readable way complex logic and arithmetic operations for PLCs.

At the same time, recent LLMs like GPT-4 [5], PaLM [25], and LLaMA [108] have brought a paradigm shift in the field of Artificial Intelligence. These have been trained on large sets of natural language as well as programming code and have proven exceptional abilities in tasks such as code generation [23], summarization [103], and error correction [50]. Interest in using LLMs to generate domain-specific code (such as automation scripts and PLC code) has been prompted by the success of LLMs in general purpose code generation.

Despite growing interest in applying LLMs to code generation and analysis, the lack of high-quality, large-scale ST datasets has hindered progress in this field. PLC code is often proprietary and safety-critical, which severely limits public data availability. Companies are reluctant to share operational control programs due to intellectual property and reliability concerns and only a few open-source libraries (e.g., OSCAT) exist, which cover limited functionality and are not readily usable for training robust models. As a result, general-purpose code LLMs (trained mostly on mainstream languages like Python/C++) struggle to generate valid PLC code, since they have not seen enough domain examples. Recent work confirms that state-of-the-art LLMs (ChatGPT-3.5, ChatGPT-4, Bard, Code Llama, StableCode, and Platypus2) often fail to produce correct PLC programs without special handling [109].

To address this data scarcity, we propose a synthetic dataset generation pipeline that produces realistic, diverse, and buildable ST code at scale. Our approach leverages LLM to generate control logic programs across a wide range of automation scenarios, validates them using industrial compilers and iteratively refines them through testing frameworks. By filtering out non-compiling or incorrect samples and continuously improving generation prompts, the pipeline can yield a large corpus of high-quality ST code suitable for fine-tuning an LLM. The resulting dataset aims to enable applications such as code scaffolding, automated code review, and documentation

in the industrial automation domain. In summary, this work contributes a methodology to bootstrap domain-specific LLMs for PLC programming by *synthetically* generating the data they need.

4.2 Background and Related Work

4.2.1 Structured Text and IEC 61131-3

Structured Text (ST) is one of the five programming languages defined in the IEC 61131-3 standard for PLCs. Unlike graphical languages (Ladder Diagram, Function Block Diagram, etc.), ST is a text-based language with syntax vaguely resembling Pascal or Ada. It supports imperative constructs such as variable assignments, loops, conditional statements (IF/THEN/ELSE), and user-defined function blocks, making it suitable for implementing complex control algorithms.

As shown in Listing 4.1, the function block encapsulates basic arithmetic operations. The main program in Listing 4.2 demonstrates its usage within a loop and conditional control structure. ST code is executed cyclically or on events within the real-time PLC runtime environment. Key features include the ability to interface with hardware I/O through memory-mapped variables. Because ST is designed for industrial control, programs must adhere to strict timing and safety requirements and typically undergo thorough validation and testing.

Despite its importance in industry, ST has a relatively small public footprint in terms of accessible source code. The standard itself is proprietary and most code is locked in vendor-specific repositories or project files. A few community-driven projects, such as the OSCAT open library, provide reusable ST components, but these are limited in scope. As a consequence, researchers have noted a lack of open datasets or benchmarks for PLC programming languages, which hampers the application of data-driven machine learning techniques in this domain. This motivates the creation of synthetic corpora and benchmarks to stimulate research. Our work specifically targets ST to fill this gap.

```
FUNCTION_BLOCK FB_Calculator

VAR_INPUT
    num1 : INT;
    num2 : INT;
    op   : STRING[5]; // 'ADD', 'SUB', 'MUL', 'DIV'
END_VAR
VAR_OUTPUT
    result : REAL;
END_VAR
VAR
    temp : REAL;
END_VAR

// Function Block Logic
IF op = 'ADD' THEN
    result := num1 + num2;
ELSIF op = 'SUB' THEN
    result := num1 - num2;
ELSIF op = 'MUL' THEN
    result := num1 * num2;
ELSIF op = 'DIV' THEN
    IF num2 <> 0 THEN
        result := num1 / num2;
    ELSE
        result := 0.0; // handle divide by zero
    END_IF
ELSE
    result := 0.0; // default
END_IF

END_FUNCTION_BLOCK
```

Listing 4.1: Example of User-defined Function Block for basic arithmetic operations in Structured Text.

```
PROGRAM Main
VAR
    i       : INT := 0;
    sum     : INT := 0;
    avg     : REAL := 0.0;
    numbers : ARRAY[1..5] OF INT := [10, 20, 30, 40, 50];
    calc    : FB_Calculator; // Instance of function block
END_VAR

// Calculate sum using FOR loop
FOR i := 1 TO 5 DO
    sum := sum + numbers[i];
END_FOR;

// Calculate average
avg := sum / 5.0; // ensure REAL division

// Call the user-defined Function Block
calc(num1 := 15, num2 := 3, op := 'MUL');

// Conditional example
IF avg > 25.0 THEN
    // WHILE loop to decrement result until <= 50
    WHILE calc.result > 50.0 DO
        calc.result := calc.result - 10.0;
    END_WHILE;
ELSE
    calc.result := calc.result + 10.0;
END_IF;

END_PROGRAM
```

Listing 4.2: Example of Main program using the user-defined Function Block in Structured Text.

4.2.2 Synthetic Code Dataset Generation

Synthetic data has become a key enabler for training LLMs in domains where real data is scarce or sensitive. In natural language processing and general programming, researchers have used LLMs themselves to generate additional training examples, a strategy that can boost performance in low-resource settings. Prior work has explored synthetic code datasets for languages like

Python and SQL, often using prompt-based generation or domain-specific grammars to create new code samples. For example, techniques such as self-instruction or automated code generation with unit test feedback have been applied to produce synthetic Python functions and accompanying tests. These approaches leverage the fact that code has a clear notion of correctness, which can be checked automatically and used to filter the generated data.

However, few efforts have targeted industrial languages like ST. One reason is the complexity of the domain and the tools required to validate such code (e.g., PLC-specific compilers and runtimes). Another reason is the specialized knowledge required to craft meaningful control programs. Our pipeline builds on ideas from synthetic data generation in general programming, such as prompt-based generation and execution feedback, but adapts them to the PLC context. By automatically compiling generated code and running PLC unit tests, we ensure that the synthetic data is not only syntactically correct but also exhibits plausible runtime behavior. This methodology is, to our knowledge, the first to be applied at scale for Structured Text.

4.2.3 LLMs in Niche Domains

Large Language Models can be fine-tuned or adapted to specialized domains to improve their performance on domain-specific tasks. This has been demonstrated in areas such as biomedicine, law, and scientific literature. For instance, domain-adapted models like SciBERT [14] were pretrained on a large corpus of scientific text, yielding better results on downstream scientific NLP tasks compared to baseline models. Similarly, BioBERT [64] and LegalBERT [21] are BERT variants tuned on biomedical and legal corpora, respectively, which significantly outperform general models on tasks in those domains. These examples illustrate that providing an LLM with in-domain data (either through pre-training or fine-tuning) can inject important domain knowledge and vocabulary, leading to more accurate and contextually appropriate outputs.

In the realm of programming, specialized code models (CodeLLMs) have emerged for mainstream languages (e.g., CodeLlama, StarCoder). Yet for niche programming languages like those in IEC 61131-3, a major challenge is the lack of training data. Recent studies highlight that for low-resource programming languages or proprietary domains, data scarcity severely limits its model effectiveness. To overcome this, researchers have begun creating synthetic or augmented datasets. For example, specialized PLC code generation frameworks have fine-tuned models on small curated sets of ST code or used techniques like Retrieval-Augmented Generation to inject domain

knowledge[70]. Fine-tuned CodeLlama on a limited ST codebase improved the success rate of PLC program generation with an iterative verification loop. These efforts reinforce the notion that domain-specific data — even if synthetic — can substantially enhance an LLM's capabilities in specialized domains.

4.2.4 Existing ST Code Repositories

- **OpenPLC Project**: An open-source initiative that provides a platform for PLC programming, including ST support. It includes a small library of example programs but is not comprehensive.
- **OSCAT Library**: A community-driven collection of reusable function blocks and routines for ST programming. It covers utility functions but lacks extensive documentation and metadata.
- **PLC-BEAD**: A dataset that pairs PLC binaries with their ST source code for security analysis. It contains around 700 programs [3].
- **PLCopen**: An organization that provides a range of resources for PLC programming, including function block libraries and guidelines for ST programming. Their resources are often used as a reference in the industry.
- **TcOpen**: An open-source project that provides a framework for PLC programming in ST, including a small codebase focused on specific use cases. It is not comprehensive but serves as an example of modern ST programming practices.
- **Vendor Example Libraries**: Many PLC vendors provide example code libraries for their platforms, but these are often proprietary and not openly available for training purposes.
- **TcUnit**: An open-source unit testing framework for TwinCAT that allows developers to write and run tests for their PLC programs. It is often used in conjunction with ST code but does not provide a dataset of ST programs itself.
- **GitHub Repositories**: Some developers share their ST code on GitHub, but these repositories are often small and focused on specific projects or examples. They do not provide a large-scale dataset suitable for training LLMs.

- **CodesysForge**: A community-driven platform that hosts various resources for PLC programming, including ST code examples. However, it is not a comprehensive dataset and often lacks quality control.

To train a large language model or even fine-tune a medium-sized model, a dataset orders of magnitude larger is desirable. By generating a synthetic dataset from scratch, we avoid the legal and practical barriers of collecting real industrial code. Our generated dataset, comprising on the order of 2,000 programs, far exceeds the size of any existing ST repository. Moreover, each synthetic sample comes with rich annotations (e.g., category labels and test cases) that are generally not available for real industrial code. We hope that releasing this synthetic corpus will bootstrap further research and applications in this area, complementing the small real-world datasets that do exist.

4.3 Design Goals and Requirements

In designing the synthetic dataset and the generation pipeline, we established several criteria and requirements to ensure the usefulness of the data for LLM fine-tuning:

- **Code Quality**: All generated programs must be syntactically correct (no compiler errors) and semantically non-trivial. This means the code should compile successfully under an IEC 61131-3 compliant compiler and implement logically plausible control behavior (not just random or dead code). We enforced this by integrating a compiler in the loop and filtering out any code that fails to build.

- **Construct Coverage**: The dataset should collectively cover the breadth of language constructs and PLC programming patterns. This includes basic constructs (loops, conditionals), use of timers and counters, arithmetic and signal processing logic, state machines (e.g., sequential function charts implemented in ST), and invocations of function blocks (both standard library and custom). Ensuring construct coverage helps the fine-tuned LLM not only memorize patterns but also understand the full syntax and features of ST.

- **Pattern Diversity**: We aim for a diverse set of programs so that the model does not overfit to a narrow style. Programs in the dataset vary in complexity (from tens to thousands of lines), in structural patterns (straight-line logic vs. event-driven branching) and in application domain (motion control, process control, etc.). We achieved diversity by

using a wide range of prompt templates and by injecting variability through random parameterization of the prompts. We also perform de-duplication and enforce uniqueness checks to avoid repetitive samples.

- **Metadata Support**: Each code sample is accompanied by metadata that can aid downstream use. This includes the high-level category (domain) of the program, a subcategory or problem description, and the initial prompt that led to its creation. Such metadata is useful for analysis, allowing one to filter or select programs by type, and could also be used to condition the LLM (for example, providing the category as part of the input context during fine-tuning). In addition, we include the expected output or behavior description (when available) which can serve as a natural language annotation for the code.

- **Build Compatibility**: The code must be not only syntactically correct in theory, but also practically buildable in a representative industry environment. We target the TwinCAT 3 development environment (a popular IEC 61131-3 platform by Beckhoff Automation) for validation. All code is wrapped in a TwinCAT project structure (with necessary boilerplate) and compiled with the TwinCAT ST compiler to ensure 100% compliance with real-world standards and library references. Any non-conforming code is corrected or eventually rejected.

- **Testability**: To support robust model training and evaluation, each program comes with a set of unit tests. We use the open-source TcUnit framework (a unit testing framework for TwinCAT/ST) to write test function blocks that call the program or function under test with various inputs and assert expected outputs. The presence of tests allows us to gauge semantic correctness of the program (does it do what the specification intended) and will enable fine-tuning the LLM not just on code but also on the relationship between code and its functional requirements (as captured in tests and assertions). Test cases are also crucial for mutation testing (described below), which we use to measure test coverage and program robustness.

By meeting these design goals, the resulting dataset is intended to be a comprehensive training resource that reflects realistic and varied industrial control programs, rather than toy examples. Next, we describe the pipeline that generates the dataset to meet the above requirements.

4.4 Synthetic Dataset Generation Pipeline

Figure 4.4.1 provides an overview of our synthetic data generation pipeline. The process is divided into several stages, each responsible for creating or validating a particular aspect of the dataset. The pipeline is designed to be mostly automated, with minimal human intervention, so that it can scale to tens of thousands of programs. This section briefly explains the details of each component of the pipeline.

4.4.1 Problem Scaffolding and Prompt Design

The pipeline begins with a problem scaffolding phase. We first defined a taxonomy of control problems to ensure coverage of diverse industrial scenarios. Top-level categories include *Motion Control* (e.g., motor control logic, PID loops), *Process Control* (such as temperature regulation, sequencing in chemical processes), *Safety Interlocks*, *Human-Machine Interface (HMI) logic*, and others. Within each category, we enumerated several problem descriptions. For example, under motion control we might have subcategories like "Conveyor belt with start/stop and emergency stop", "Robot arm pick-and-place sequence", etc. Each problem description is essentially a specification of a control task.

In many cases we employed a few-shot prompt: we supplied one or two small examples of ST code (unrelated to the target task) along with explanations, to prime the model on the style and syntax of ST. This technique was used to improve the consistency and correctness of the LLM's output from the very first generation step.

We found that including even a tiny example greatly reduced the incidence of syntax errors and helped the model output well-structured code. The prompt templates also often ended with explicit instructions like "Ensure the code compiles without errors" to remind the model to be careful with syntax.

4.4.2 LLM-Based Code Generation

Azure OpenAI GPT-4 service is employed as a primary code generation engine (though the pipeline can work with other LLMs as well, given the prompts). For each prompt from the previous stage, the LLM generates an ST *project* as its response. The generation is done with a temperature parameter that injects some randomness to avoid deterministic repetition, but we keep it moderately low to maintain focus and correctness. In cases where

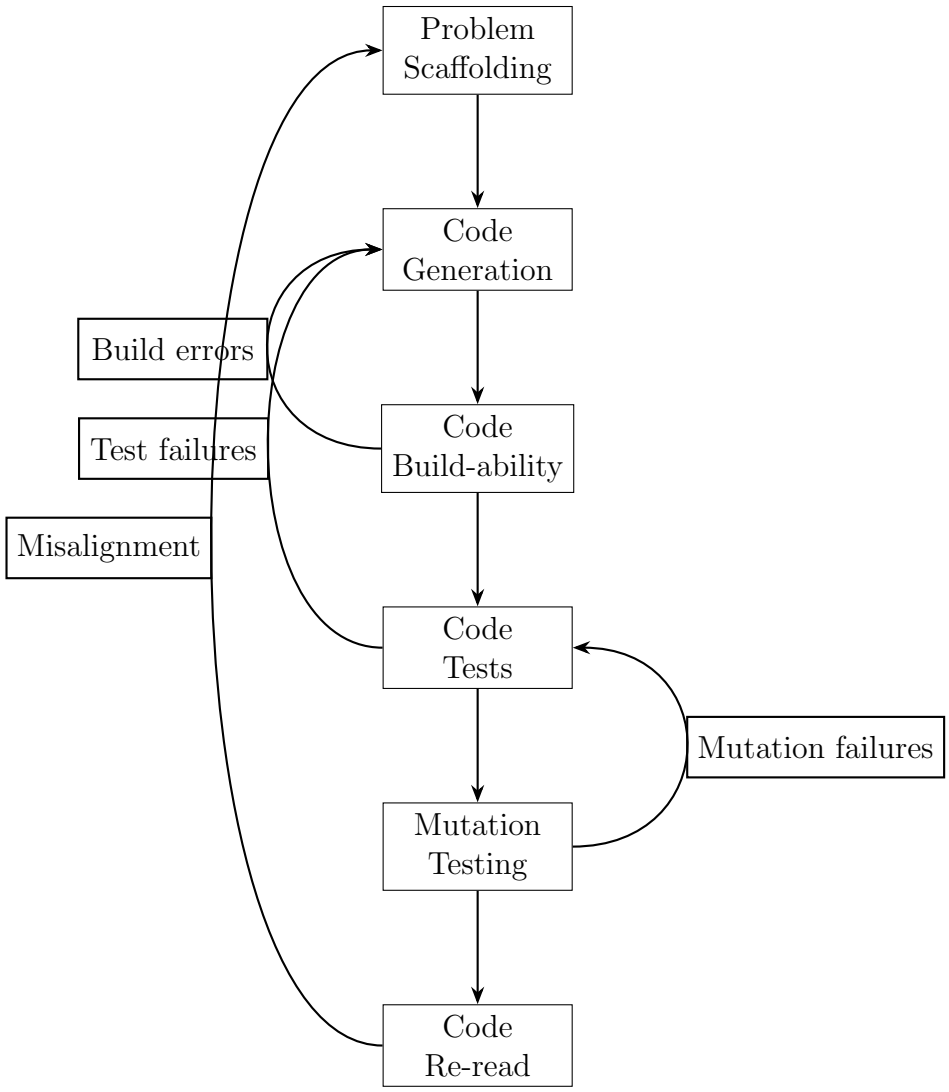


Figure 4.4.1: Overview of the synthetic Structured Text code generation pipeline. The system iteratively generates ST programs using an LLM guided by prompts, then validates them through compilation and testing in a TwinCAT environment. Failed cases are automatically corrected via LLM feedback loops and the final set of programs is subjected to mutation testing for evaluating test coverage.

we wanted multiple variants of a solution, we could sample the LLM multiple times with the same prompt (perhaps with slight rephrasing or different random seeds).

We ask GPT-4 to create ST *projects* using a specific `format_response` instruction, which ensures the model follows a user-provided JSON schema, which is a valid port to json of a TwinCAT project XML structure: in fact, TwinCAT (like most PLC IDEs) uses an XML project format which we can programmatically manipulate and then compile using an automation interface.

At this stage, we treat the code as a candidate. We do not yet consider it part of the final dataset; it must pass through validation steps. But we do keep the log of all generated code and associate it with the prompt that produced it. This traceability is useful for analysis and for iterative prompt refinement (if many generations fail, we adjust the prompt templates accordingly).

4.4.3 Code Build and Validation

Once a piece of code is generated and embedded into a project, the next step is to verify its **buildability**. We use the TwinCAT automation interface, provided by the underlying Visual Studio instance, to attempt a compile of the ST project. Programmatically, this involves loading the project, invoking the build, and capturing any compiler output or errors. This step is crucial: any syntax errors, type mismatches, or missing references will be caught here. If the code compiles successfully, it moves on to the next stage. If it fails to compile, we capture the error messages for analysis.

A key innovation in our pipeline is automated code correction via the LLM. If a compilation error occurs, we feed the error message and the offending code back into the LLM in a refinement prompt.

Often, the model can identify the mistake (e.g., a variable was used without declaration, or a mismatched data type) and suggest a corrected code. We then replace the original code with the corrected one and try to compile again. This iterative loop continues until the code compiles or a maximum number of attempts is reached. In our experiments, we found that the majority of compilation issues could be fixed in a couple of refinement iterations (since they were usually small oversights). This approach is in line with the concept of using LLMs to "learn from failure" by feeding back errors and letting the model correct them.

It's worth noting that because our initial prompts and few-shot examples were carefully designed, a large fraction of generated programs compiled on the first try. The automated correction loop raised the overall compilation

success rate to over 99%. The remaining failures were often due to more complex issues, which we handled by dropping that sample.

By the end of this stage, we have a set of syntactically valid ST programs. Each has been verified to build in TwinCAT, meaning it conforms to IEC 61131-3 rules and TwinCAT's implementation thereof. The code is ready to be tested for functional correctness.

4.4.4 Automated Test Case Generation

For each program, we generate corresponding unit tests using the LLM. Our approach is to prompt the LLM to produce a TcUnit test function block given the context of the program. We supply the code (or a description of its functionality) and ask for a test suite that covers the main logic paths. For example, for a program controlling a traffic light, the prompt would instruct: "Generate a TcUnit test function block that instantiates the traffic light controller and simulates pressing the pedestrian button at different times, then asserts that the light sequence follows the specification."

After generating tests, we integrate them into the TwinCAT project and compile again. A test not compiling could indicate the test code had mistakes, which we then correct similarly via an LLM refinement if needed.

With compiled test binaries, we execute the tests using the TcUnit framework (which can run within a TwinCAT runtime or a simulation environment). The tests exercise the code and produce a simple report of passed/-failed assertions. If any test fails, that indicates the program did not behave as expected for the scenario. Such a failure triggers a review: either the test is too strict or the code has a logical bug. We choose to address it by further refining the code using the LLM, this time guided by the failing test information. For example, if a test expected output X but got Y, we prompt the LLM to adjust the code accordingly (or rarely, adjust the test if the expectation was wrong).

This iterative test-driven refinement is analogous to a human debugging cycle and greatly increases the semantic reliability of the generated programs. Only programs that eventually pass all their tests are kept. By enforcing that each program comes with a passing test suite, we ensure a degree of semantic correctness.

4.4.5 Mutation Testing and Quality Assurance

As a final layer of quality control, we perform mutation testing on the programs. Mutation testing involves introducing small changes (mutations) into the program code to see if the test suite can catch the errors. It is a way of

measuring the thoroughness of the tests. If a mutated program still passes all the tests, it suggests the tests are not sensitive to that change, possibly indicating insufficient coverage.

We implemented several mutation operators relevant to ST:

- Negating boolean conditions (e.g., replace 'IF condition THEN' with 'IF NOT condition THEN').
- Changing comparison operators (e.g., '>=' to '>', '==' to '<>').
- Altering constant values by a small amount (e.g., if a timer preset is 5 seconds, mutate to 4 or 6 seconds).
- Removing or bypassing a function call.

For each program, we generate a number of mutant versions (typically 2-4 mutants per program, depending on program size and complexity). Each mutant is compiled and the test suite is run against it. We record whether the tests fail (which is good, meaning the tests caught the injected fault) or still pass (which means the tests missed it). The percentage of mutants that are detected (killed) by the tests is the mutation score for that program.

A high mutation score (close to 100%) gives confidence that the tests thoroughly exercise the program's logic. A lower score suggests some logic might not be fully tested. In our pipeline, if a program had a mutation score below a threshold (e.g, 80%), we revisited the test generation step to add more tests and/or flagged the program for review.

Through mutation testing, we aim to ensure that our dataset not only contains code, but code paired with robust tests.

4.4.6 Iterative Refinement and Dataset Expansion

The entire process above can be iterated with improved prompts and additional problem scenarios. In our project, we ran multiple batches of generation. After an initial batch of about 100 programs, we analyzed the results, identified common shortcomings and then updated the prompt templates or few-shot examples to address those. We also expanded our taxonomy of tasks to cover any gaps noticed.

4.5 Quality Evaluation Metrics

To assess the quality of the synthetic dataset, we define a set of evaluation metrics, drawing from software quality metrics and dataset statistics. These

metrics help demonstrate that the data meets the design goals outlined earlier. Below are the key metrics and how we evaluate them:

4.5.1 Syntactic Correctness

This is measured as the percentage of generated programs that compile successfully without errors. Since our pipeline filters out non-compiling programs, the final dataset ideally has 100% syntactic correctness. We log the compilation success rate at initial generation vs. after the automated fixes. For example, initial generation might yield around 85% compilable code, which our refinement process raises to over 99.5%. We also keep track of the types of compiler errors encountered to inform future prompt improvements.

4.5.2 Mutation Score (Test Coverage)

For each program, we have the percentage of mutants killed by its test suite. We aggregate this across the dataset (e.g., average mutation score, median, distribution). This reflects how thorough the tests are and, by extension, how well-specified the program behavior is. An average mutation score higher than 80% would indicate most programs have strong test suites. We consider this important because it ensures the dataset can support training not just code generation but also verification and debugging capabilities of an LLM. We also track the common reasons for survivors (e.g., equivalent mutants or untested branches).

4.5.3 Dataset Size and Coverage

Although not a quality metric per se, we report the final dataset size (number of programs, total lines of code) and how they break down by category. This demonstrates that we indeed have the scale and breadth we aimed for. We define coverage here as covering the intended categories and problem types in our taxonomy. Ideally, each category from Section 4.4.1 has a substantial number of instances in the dataset. We also verify that no single category dominates excessively to avoid biasing the model toward one type of problem.

These metrics collectively give a comprehensive picture of the dataset. In the next section, we present the results of applying these metrics to our generated dataset.

4.6 Results

After running the synthetic data generation pipeline with iterative improvements, we obtained a dataset of ST programs that meets our design criteria. In this section, we summarize the key results and characteristics of the dataset.

4.6.1 Dataset Statistics

Our final dataset consists of approximately **2,000** unique ST projects at the moment. These projects span over **10** top-level categories of control tasks and dozens of subcategories. The entire corpus comprises roughly thousands lines of ST code in total. All projects are accompanied by corresponding TcUnit test blocks.

4.6.2 Limitations and Threats to Validity

Despite the promising results, our approach has several limitations:

- **Lack of Real-world Validation:** We have not yet deployed a model fine-tuned on this synthetic data to see how it performs on truly real-world tasks (e.g., solving a problem from an industrial partner). The ultimate test would be if a fine-tuned model can assist engineers in writing ST code or catching bugs. It's possible that certain nuances of real projects (timing constraints, integration with hardware, etc.) are not fully captured by our synthetic scenarios.
- **Coverage of Company Specific Libraries:** As mentioned, we largely stuck to standard TwinCAT ST and basic library functions. Real PLC code often uses complex libraries (for motion control, fieldbus communication, etc.). This is a gap that needs to be filled either by incorporating vendor documentation into a RAG system or by expanding the dataset to include vendor-specific examples.
- **Potential Biases in Synthetic Data:** The data reflects the behavior of the underlying LLM (GPT-4 in our case) and our prompt design. If the LLM had hidden biases or blind spots about certain control logic, those could manifest in the dataset.
- **Quality of Tests Could be Artificially High:** While high mutation scores are good, they mean the tests are very tailored to the code. In the real world, tests are often incomplete. A model trained on our data

might expect that every piece of code has a full test suite, which is rarely true in practice. This could lead to a model that, when given an unseen problem without tests, might not know how to proceed (though one could argue we can have the model generate tests too). Additionally, the model might learn to rely on comments or test cues in the training data that would not be present when it's generating code in isolation. Careful fine-tuning strategies (or adding variety, such as training also on some samples with no tests) might be needed to avoid over-reliance on test presence.

- **Compute and Cost:** Generating thousands of programs with an LLM and validating them is not cheap. We used significant computation resources and OpenAI API credits. While this is a one-time cost (and arguably much cheaper than gathering equivalent data from industrial partners, which might be impossible), it could be a barrier for others to replicate or extend the dataset. However, one could generate a smaller dataset or use open-source models (with perhaps lower quality) if cost is an issue.

- **Intellectual Property:** If the LLM is not trained completely "in house", there is the possibility that the code generated by the LLM violates the IP of some company/entity whose code was used in the training phase.

4.7 Conclusion

We presented a scalable pipeline for generating and validating a synthetic Structured Text (ST) code dataset for the purpose of fine-tuning large language models in the industrial automation domain. By leveraging LLMs in conjunction with compiler feedback, testing, and mutation analysis, we demonstrated that it is possible to create a high-quality corpus of PLC programs complete with specifications and test cases. This addresses a critical bottleneck in applying AI to industrial code: the lack of publicly available training data.

Our synthetic dataset enables training domain-specific LLMs that better understand PLC programming tasks. Fine-tuning an LLM on this data has the potential to drastically improve its ability to generate correct and industry-applicable code, as well as to reason about control logic.

Chapter 5

Licensing and Innovation Management

5.1 Licensing

The increasing integration of AI technologies in the industrial settings also raised serious legal, ethical, and operational concerns, especially in regard to implementing Large Language Models. LLMs are already applied in different industries to automate work, make better decisions, and become more productive. The application of these models are, however, controlled by the licensing agreements that control their use, alteration, and sharing. This chapter discusses the issue of licensing that are important in the application of LLMs in an industrial setting e.g. packaging solutions and manufacturing. It classifies the type of licenses, compares the deployment models, analyzes the constraints related to industrial use cases, and provides the recommendations based on the risks and opportunities related to licensing. In grasping these aspects, companies can guarantee sustainable and compliant implementation of LLMs in their operations.

5.1.1 Types of LLM Licenses

Large Language Models are released with many different licenses, each with its own conditions and consequences to users. Three broad categories are open source licenses, proprietary licenses, and non-commercial or research-only licenses.

Open source licenses

The users of such licenses are usually free to download, alter, and share model's code and weights. Examples are the Apache 2.0 [98] and MIT licences [94]. In industrial applications usually suits permissive open source licenses because of their flexibility of use. Copyleft licences such as the GNU General Public License (GPL) [69] has even more requirements such as the need to release derivative works under the same license.

Proprietary licenses

Proprietary models such as OpenAI GPT-4 [4] or Claude[9] by Anthropic are sold through via proprietary APIs and governed by Terms of Service (ToS). These deals tend to limit local deployment, alteration, and resale of the models. Although these types of models are easy to integrate, but they reduce flexibility and pose a problem of vendor lock-in.

Non-Commercial and Research-only Licenses

Some models, including Llama 2 [108] and Llama 3 [42] by Meta, are only available to academics and researchers unless more commercial contracts are signed. Such licenses do not allow out-of-the-box industrial application and need to be negotiated over commercial deployment.

5.1.2 Deployment Models and Legal Implications

The license applicability and compliance is influenced by the deployment strategy of an LLM such as Cloud-based or on-premise.

Cloud-based (API Access)

The model entails using the LLMs through APIs provided by developers such as OpenAI, Anthropic, or Google. It is convenient, but the user is subject to Terms of Service of the provider. These models normally do not allow reverse engineering, impose data handling rules, and do not guarantee output reliability. Besides, API access can involve repetitive expenses, time latency, and the risk of data privacy.

Private hosting or On-Premise

On-premise hosting is usually favored by industries such as Coesia that deal with sensitive data. Open-source solutions such as Phi-3-mini-128k-Instruct [1] would be perfect in this case provided that they can be used

commercially. Such a solution provides more control over the flow of information, confidentiality, and system integration but needs technical skills in hosting and securing models.

5.1.3 Use Case Constraints in Industrial Settings

The use of LLMs in an industrial settings have its own peculiarities, which are not usually typical of consumer or academic ones. These limits define the way licenses have to be interpreted and applied.

Sensitivity of data

Manufacturing, Packaging solutions, Healthcare, Finance, and Automation firms are the industries that deal with sensitive data. Regulatory frameworks such as the General Data Protection Regulation (GDPR) in the EU or the Health Insurance Portability and Accountability Act (HIPAA) in the U.S. demand that data should be treated with strong privacy protection. In case an LLM is implemented through an external API, it should be checked whether user data is stored, logged, or used to further train a model.

Ownership of the Intellectual Property (IP)

The products produced by LLMs in an industrial environment, e.g. code, technical documentation, reports, or design components, can be intellectual property. The ownership of such outputs may be restricted or made complicated by some proprietary licenses. The businesses should assess whether the output is of the user or the model provider.

Fine-tuning and Customization

Model fine-tuning with proprietary data allows the model to become domain-specific, although it can generate a derivative work with some licenses. As an example, a RAIL license on a model may not allow commercial use of any fine-tuned version of it, should it contribute to surveillance or the development of autonomous weapons. Before such model adaptations, licensing restrictions will have to be studied.

Availability and Reliability

Industrial tasks can rely on the LLMs to make real-time decisions or even in some industrial critical tasks. It means that models must possess high uptime guarantees and evident fault recovery procedures. In cases where

models are accessed through external APIs have to be reviewed to achieve acceptable levels of service continuity.

Explainability and Compliance Audits

In regulated industries, it is often necessary to have the capability to explain model outputs. The model and the decision-making processes should be inspectable and auditable under licensing. Some proprietary solutions may limit visibility into the model's architecture or training data, making explainability difficult to achieve.

5.1.4 Risk Management in License Compliance

Licensing LLMs involves legal, business, and operational risk management considerations. Violation of license terms may have serious consequences that include legal actions as well as the disruption of business.

Legal Risks

Non-observance of the conditions of the license may lead to cease-and-desist letters, fines, or compulsory termination of a product or service. As an example, re-distributing a model or a fine-tuned derivative without appropriate attribution, or under a license more permissive than the original license can violate the original license.

Security and Compliance Risks

LLM deployments in the Cloud pose the danger of sending sensitive industrial data outside of organization's secured environments. Unless the licensing conditions contain data retention, encryption, or auditability clauses. The enterprise can unintentionally infringe the data protection laws.

Operational Risks

The work processes in industries must be very reliable. In case the LLM service provider modifies their terms of the license (e.g. fees or usage restrictions), this may upset operations. In the same way, no support or maintenance of the open-source models may pose some long-term dependency risks.

Reputational and Ethical Risks

Training a model with copyrighted or ethically dubious information may hurt the reputation of a company. The licensing terms with aspects of ethical usage, e.g. in RAIL licenses, can be protective mechanisms, as well as restrictive in the operation.

5.1.5 Comparative Case Studies

In order to have a closer insight into the impact of license type on the industrial deployment, this section provides a comparison of state-of-the-art LLMs on various dimensions. In Table 5.1.1, the effects of license selection on deployability, flexibility, and adherence to privacy requirements are listed. The open-source models are more flexible, whereas proprietary APIs are easier to access at the cost of limitations.

5.1.6 Recommendations for Industrial Deployment

The best practices of the deployment of LLMs in the industrial environment based on the comparative analysis and risks identified are as follows:

- Apply permissive open-source frameworks such as Phi-3-mini when complete control, visibility, and customization is needed.
- Review the legal and ethical provisions contained within any open-source license, especially where such open-source is used in regulated or sensitive fields.
- Do not over rely on proprietary APIs unless the Service Level Agreements, data policies, and commercial terms are completely aligned to business demands.
- Conduct legal reviews as well as license audits prior to the incorporation of any model into the production line.
- Keep fallback models or hybrid deployments in order to avoid service interruption due to change in licensing.

Record and track all communication with the LLM in order to be compliant and explainable. Enterprise negotiation of contracts with vendors can be done when commercial APIs are used in order to achieve long term stability. Licensing is one of the core elements of successful and sustainable application of Large Language Models in the industrial environment. The kind of license

Table 5.1.1: Code Generation Large Language Models (2020-2025)

Model	Developer	Release Date	License	Commercial Use	Fine-tuning	Notes	Source
Codex	OpenAI	2021-08	Proprietary	Yes (API)	No	Powers GitHub Copilot; multi-language support; fine-tuning unavailable	[23]
CodeT5	Salesforce	2021-09	BSD 3-Clause	Yes	Yes	Encoder-decoder transformer for code generation and understanding	[112]
CodeGen	Salesforce	2022-03	Apache 2.0	Yes	Yes	Autoregressive transformer supporting Python, Java, JavaScript	[80]
PolyCoder	CMU	2022-05	MIT	Yes	Yes	Open-source code generation model trained primarily on C/C++	[118]
CodeBERT	Microsoft	2020-09	MIT	Yes	Yes	BERT-based model for code search and document generation	[37]
GraphCodeBERT	Microsoft	2020-09	MIT	Yes	Yes	Incorporates data flow info to improve code understanding	[44]
PLBART	California & Columbia University	2021-03	MIT	Yes	Yes	code summarization in the English language, code generation, and code translation in seven programming languages	[7]
InCoder	Facebook AI Research	2022-04	CC-BY-NC 4.0	No	Yes	Autoregressive model focused on code completion	[38]
SantaCoder	BigCode	2022-12	BigCode OpenRAIL-M	Yes with restrictions	Yes	Open-source model trained on permissive-license code	[8]
CodeGen-2	Salesforce Research	2023-05	Apache 2.0	Yes	Yes	Larger, improved version of CodeGen with strong infilling capabilities	[81]
CodeGeeX	Z.ai and THUKEG	2022-09	Apache 2.0	Yes	Yes	13B parameter multilingual code model	[125]
StarCoder	BigCode	2023-05	BigCode OpenRAIL-M	Yes with restrictions	Yes	15B param model trained on permissively licensed code	[66]
Code LLaMA	Meta AI	2023-08	Meta License	Limited	Yes	Code-specialized version of LLaMA	[93]
StarCoder-2	BigCode	2024-02	BigCode OpenRAIL-M	Yes	Yes	small model StarCoder2-3B, outperforms other Code LLMs of similar size on most benchmarks	[71]
CodingGenie	Microsoft Research	2025-03	Apache 2.0	Open Source Tool	Yes	autonomously provides suggestions, ranging from bug fixing to unit testing, based on the current code context and allows users to customize suggestions by providing a task description and selecting what suggestions are shown	[124]
CodeFuse-13B	Ant Group	2023-10	Apache 2.0	Yes	Yes	Multilingual code LLM	[35]
GPT-NeoX-20B	EleutherAI	2022-04	Apache 2.0	Yes	Yes	largest dense autoregressive model that has publicly available weights at the time of submission and powerful few-shot reasoner	[17]
CodeT5+	Salesforce AI Research	2023-05	BSD 3-Clause	Yes	Yes	Family of encoderdecoder LLMs for code in which component modules can be flexibly combined to suit a wide range of downstream code tasks	[113]
GPT-4	OpenAI	2023-03	Proprietary	Yes (API)	No	State-of-the-art code generation via API	[4]
GPT-4o	OpenAI	2024-10	Proprietary	Yes (API)	No	Fast and cheaper GPT-4 variant for code	[55]
AlphaCode	DeepMind	2022-02	Proprietary	No	No	Competitive programming-focused code model	[68]
PaLM	Google Research	2022-04	Proprietary	Limited	No	Multilingual tasks and source code generation	[25]
Unixcoder	Microsoft	2023-03	MIT	Yes	Yes	unified cross-modal pre-trained model for programming language	[45]
Wizardcoder	WizardLM Team	2023-06	Apache 2.0	Yes	Yes	Empowers Code LLMs with complex instruction fine-tuning, by adapting the Evol-Instruct method to the domain of code	[74]
OpenCoder	INF & Multimodal Art Projection	2024-11	MIT	Yes	Yes	open source model serves as an "open cookbook" for the research community	[53]
CodeGemma	Google	2024-04	Proprietary	—	—	state-of-the-art code completion model designed for fast code infilling and open-ended generation in latency-sensitive settings	[105]
AlphaCodium	CodiumAI Inc.	2024-	GNU	Yes with restrictions	Yes	Test-based multistage code-oriented iterative flow that improves the performances of LLMs on code problems	[91]
aiXcoder-7B	Peking University, aiXcoder	2025-01	Apache 2.0	Yes	Yes	LoRA/adapters supported	[58]
Qwen-Coder	Qwen Team and Alibaba Group	2024-11	Apache-2.0	Yes	Yes	Fine-tuning supported on Hugging Face	[54]
Qwen3	Qwen Team and Alibaba Group	2025-05	Apache-2.0	Yes	Yes	Fine-tuning supported on Hugging Face	[120]
Seed-Coder	ByteDance Seed	2025-06	MIT	Yes	Yes	Demonstrated superior performance in code generation, code completion, code editing, code reasoning, and software engineering tasks	[95]
Gemini 2.5 Pro	Google DeepMind	2025-07	Proprietary	Yes (Google Cloud API)	No	SoTA performance on frontier coding and reasoning benchmarks	[29]
Phi-3	Microsoft	2024-05	MIT License	Yes	Yes	Optimized for small devices	[1]
Phi-4	Microsoft	2024-12	MIT License	Yes	Yes	Despite minimal changes to the phi-3 architecture, phi-4 achieves strong performance relative to its size – especially on reasoning-focused benchmarks	[2]
GPT-OSS	OpenAI	2025-08	Apache 2.0	Yes	Yes	Designed as an open GPT alternative	[6]

does not only define what the model can or cannot do but also how it will fit in pre-existing systems, its data management capabilities, and how companies can expand their usage of the technology. The open-source models are more controllable and flexible, though they might be complicated legally and operationally. Proprietary APIs are easier to access and support, but offer less transparency and control. Managing these trade-offs is crucial in achieving legal, ethical, and successful applications of LLMs in the high-stakes settings. A sophisticated interpretation of licenses will enable industries to be more responsible in their innovation, reducing legal, and operational risks.

5.2 Innovation Management in Industrial Adoption of AI-Driven Code Generation Tools

Code generation with LLMs is a significant breakthrough in Software Engineering. The potential of these AI-based tools change the software development lifecycle by enabling new modes of productivity, efficiency, and creativity. For industrial automation firms such as Coesia, which develops sophisticated manufacturing and packaging solutions, the integration of these technologies comes with both opportunities and challenges as well. On the one hand, tools based on AI have the possibility to speed up the process and minimize the repetitive tasks and ennhance the modularity of the machine control software. The integration of such technologies into established workflows requires careful management of innovation. But this technological advancement should not be assessed in the technical perspective alone but also in the wider organizational, economic, and human context as well. In this regard, innovation cannot be treated as a creation of new capabilities only, but rather as a systematic management of technological change and value addition.

Innovation management [40] is an interdisciplinary field, which deals with the systematic ways in which organizations can foster, direct, and execute innovation. In the era of GenAI, and particularly in the context of the fast integration of AI and LLMs, the management of the innovation process should be treated carefully, as the introduction of new technologies into the working processes, their alignment to business goals, and their responsible and sustainable use should be addressed. Effective management of innovation entails the evaluation of organizational preparedness, identification and monitoring of key performance indicators (KPIs), risk management, regulation and ethical issues, and development of stakeholder's trust.

This section puts the discussion into the context, particularly focuses on

the development of an AI-based coding assistant. The assistant is designed with the aim of using limited computational resources to fit in industrial automation settings. One of the most important stakeholders in the industry for this study is Coesia S.p.A., a leading firm worldwide in automation and packaging solutions. It also explores the fact that the creation of this coding assistant that is driven by an LLM is not merely a technical feat but a well-organized innovation exercise as well. In this context, it examines some of the fundamental aspects of innovation management, including the level of technology preparedness and implementation capabilities of the system, change management and stakeholder involvement strategies. In doing so, this section seeks to present practical suggestions as well as theoretical consideration on how to go about the implementation of LLM-based tools within industrial contexts.

5.2.1 Drivers of Innovation in Code Generation

Deployment of code LLMs like GPT-5-Codex, GPT-4 is a revolutionary change in Industrial software engineering. These models produce syntactically and semantically correct code and enhance developer's productivity, reduce time and cost, and also reduction in errors [76]. The major factors that contribute to innovation are:

- Increase Productivity:

  When developers were put in a controlled experiment, they were able to complete their tasks 55.8% faster with GitHub Copilot [86]. The report of McKinsey & Company also indicates that generative AI tools can reduce the development time by half on typical software engineering work [33]. LLMs assist the less experienced programmers a lot. According to empirical findings, junior developers are more benefited by the use of such tools as Copilot, which allows them to write better code and achieve better learning performance [127][85].

- Rapid Prototyping:

  LLMs such as GPT-4, Agents4PLC, and LLM4PLC can be used to rapidly generate and verify PLC code in domain-specific use cases, for example, in industrial automation, thereby facilitating the accelerated development of systems and prototypes [36][70].

- Multilingual Support and Domain-Specific Support:

  Large Language Models can be designed to support multiple programming languages like those of CodeGeeX [125] and Code LLaMA [93]

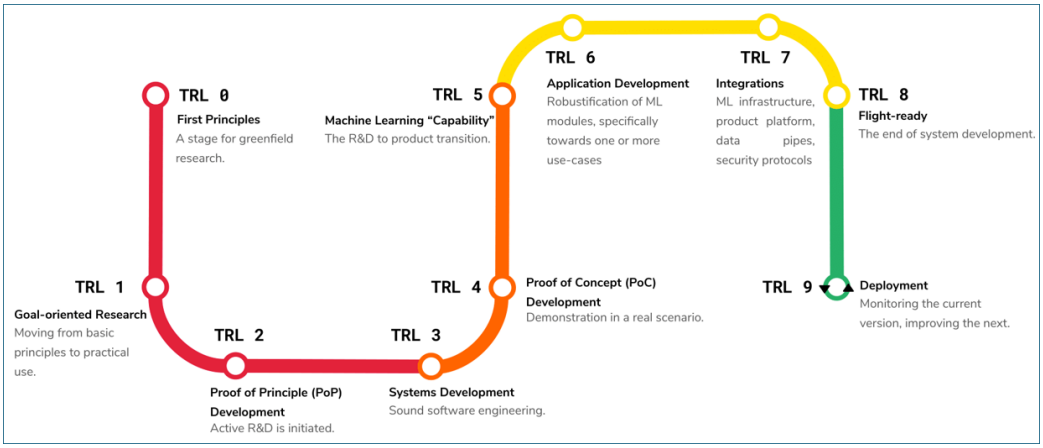


Figure 5.2.1: MLTRL spans research through prototyping, productization, and deployment[61].

that can support multilingual programming languages. Structured Text is broadly used in industrial settings. The need to reduce the development time and bugs rates in more complicated industrial software system is a driving force behind the use of AI-based code generation tools. These tools not only make workflow more efficient, improve code quality, and shorten the time-to-market, but also are strong drivers of innovation as well [70][36][117].

5.2.2 Technology Readiness and Commercialization Potential

The technology maturation framework of developing LLM-based code generation tools used in industrial automation should be structured. Machine Learning Technology Readiness Level (MLTRL) [61] is a scale used to measure the technological maturity of AI/ML innovation where a specific technology may range between TRL 1 (where basic principles have been observed) to TRL 9 (where the actual system has been proved in the operational environment) as shown in Fig. 5.2.1. In our pipeline of generating synthetic code dataset prototype makes use of Azure OpenAI GPT-4 to produce domain-specific Structured Text (ST) dataset, and is connected with TwinCAT to validate and run in PLC environment.

The system has obtained a mid-level maturity, which is between TRL 5 and TRL 6, where the generated code is validated in a laboratory setting, and they are also proven in a corresponding industrial environment. In particular, the AI-based assistant has been effectively applied to produce PLC-

compatible code, which has been checked through a closed build-validation loop in TwinCAT, which is an indicator of operational feasibility. The stages are essential to building trust in the system functionality and compatibility with the current automation toolchains.

Although general-purpose and code LLMs, including GitHub Copilot [39], Twincat Chat [13], and other AI-assistants have performed well on the task of natural language to code, there are some challenges in their direct use in industrial automation, including low-latency response requirements, deployment on low-power edge devices, developer's systems, and compatibility with legacy PLC systems [117][126]. Thus, domain-specific languages such as ST require specialized adaptation to function effectively in their respective domains, and are disproportionately under-represented in open-source datasets and model pretraining corpora. The assistant has some distinct value propositions in terms of commercialization such as:

- Market Opportunity:

  Structured Text is a popular IEC 61131-3 language in industrial automation, and few open-source tools or LLM-based assistants are available to work with it in particular. This is a product that has a market need.

- Reliability in Safety-Critical Domains:

  The fact that such systems are provided with the real-time validation and code-checking mechanism helps to assure that the software works as intended, copying with safety-critical areas, which is crucial in such domains as manufacturing and packaging solutions.

- Edge-Compatible Architecture:

  Narrowing the requirements to compute, it allows deploying the system in developer's PCs and edge devices as opposed to the Cloud-intensive options. It is in line with the trend in edge-adapted LLMs and on-device model compression [122][126].

5.2.3 Measuring Innovation Impact and Value Addition

Measuring the effectiveness of the LLM-based tools pre-supposes the establishment of clear Key Performance Indicators (KPIs), such as the time saved during the development process, number of errors decrease, the rate of successful users adoption, and the savings associated with it. Quantitative information as well as qualitative feedback provide information as to the ef-

fectiveness of the innovation. Mapping these metrics to the business goals make sure that the implementation of AI will be reflected in the actual value in the form of a more reliable system, industrial standards compliance, and efficiency in operations.

5.2.4 Barriers to Innovation and Mitigation Strategies

New technologies are not always welcomed especially in strictly regulated firms. These barriers are particularly relevant in high-tech industrial contexts where machinery, digital systems, and client-specific requirements intersect. Key barriers include trust, integration complexity, and computation limits.

Trust

Engineering professionals might be lacking trust in the AI generated code, particularly when it is applied in safety-critical applications.

Integration Complexity

New tools are not easily integrated with legacy software and hardware systems and this can be expensive.

Computation Limits

Most industrial settings cannot accommodate the use of good computing resources, thus the traditional LLMs cannot easily be implemented.

The obstacles to AI-adoption are technical challenges to integrating with legacy industrial systems, limited computing resources, organizational resistance, and ethical issues. Some of the strategies to address them are educational programs to develop AI literacy, pilot projects to prove their value, and open policies covering ethical and workforce implications. The cooperation between AI developers and end-user is central to the customization of solutions as per the practical needs and the development of trust.

This study takes care of these obstacles in the following ways:

- Transparency and Traceability:

  All code generation processes and checks are kept in a log as well as are auditable.

- Validation Integration:

  The build-check performed by the TwinCAT only presents the user executable code.

- Low-Resource Optimization:

  Parameter-Efficient Fine-Tuning technique such as LoRA, along with quantization techniques enable fine-tuning large models on consumer-grade hardware. Furthermore, the education of the stakeholders, documentation, and phased implementation plans are suggested to raise the trust and integration gradually.

Coesia can sustain innovation while maintaining operational reliability and client confidence. These approaches ensure that new technologies are not only advanced but also practical and adoptable across diverse industrial settings.

5.2.5 Alignment with Broader Innovation Frameworks

The methodology and implementation of this project align with several well-established innovation frameworks:

Open Innovation

Chesbrough [24] also coined the term open innovation, where he stressed on the need to exploit external ideas and paths to market as well as internal ones. This project is an example of open innovation since it incorporates external AI models, such as Azure OpenAI GPT-4, Phi-3-mini-128k-Instruct, and with industrial partners to validate the progress of the project.

Lean Startup

Ries [89] proposed a build-measure-learn feedback loop in order to create a product that can be responsive to the needs of the customer. The process of creating a code assistan is a part of this methodology and enables the rapid prototyping and improvement process based on the feedbacks received by the user.

Diffusion of Innovations

Rogers et al. [92] described the process of spreading innovations through certain channels over the time among the representatives of a social system. The project pre-conditions the wider adoption of the tool by the industry as the process of involvement of early adopters in Coesia will promote the effectiveness of the tool.

Responsible Research and Innovation (RRI)

Stilgoe et al. [102] described the RRI that focused on the necessity of the research and innovation processes to be ethically acceptable, sustainable, and socially desirable. RRI principles are included in this project through transparency, stakeholder involvement, and their needs alignment.

5.2.6 Case Studies and Industrial Feedback

Recent generative AI breakthroughs have resulted in the creation of several coding assistants based on Large Language Models (LLMs) including ChatGPT [82], GitHub Copilot [39], Devin AI [28], Google Gemini [41], Claude [9] by Anthropic, and TwinCAT Chat [13] by Beckhoff. These tools are different in their scope and integration levels, some of them integrated into modern IDEs and some providing full-stack assistance abilities. But no such assistant, at present can reliably produce high-quality Structured Text code of industrial automation as described in Chapter 3.2. The weaknesses of the above mentioned tools are the low quality of the code, offline inability, and inability to integrate with PLC development environments. This finding strengthens the innovation gap covered by this study and confirms the necessity of a lightweight domain-specific assistant based on LLM that is trained to work with ST programming and limited resources. There are still challenges of achieving ethical governing, dealing with autonomous behaviors, and scaling deployments on resource-constrained industrial environments. Ongoing research will focus on enhancing robustness, explainability, and broader industrial integration, with innovation management playing a pivotal role in balancing technology and organizational readiness.

Chapter 6

Results and Discussion

Chapter 4 presented a comprehensive pipeline for generating synthetic ST code Dataset at scale, addressing the scarcity of publicly available datasets for industrial automation. In this context, the synthetic dataset generation pipeline described in Chapter 4 is fully functional. However, the comprehensive set of ST problems envisioned for large-scale dataset development is not yet available. At this stage, only a small dataset has been constructed to demonstrate the pipeline's capabilities. Instead of relying solely on the synthetic dataset, Experiment 6.1.4 was conducted using a hybrid dataset that combined an open-access collection of ST programs with a representative subset of the synthetic dataset generated by the proposed pipeline.

6.1 Results Analysis

6.1.1 Experiment 1

This section presents the results of open-source code Large Language Models. I have evaluated the performance of different models on a custom ST benchmark. This experiment is guided by the objective of replicating the research of Tran et al.[109]. They benchmarked the capability of multiple general-purpose LLMs to generate ST according to IEC-61131-3. In my thesis, this experiment serves as a baseline against which I compare adaptation approaches (few-shot prompting and fine-tuning).

The experimet has been performed on the 21 representative PLC programming tasks (including mathematical functions, process control, timer, interlocks, and multi-line programs) as suggested by [109]. The chosen LLM models were Code Llama 7B, Platypus2, and StableCode.

Each model was prompted in a zero-shot manner by asking queries of

the form "Give me an IEC 61131-3 Structured Text program ..." , and the resulting code was evaluated based on three metrics,

- CodeBERTScore (F3) as the similarity of the generated code to reference implementations.
- pass@k (k = 1, 2, 3) as an approximation of the likelihood of generating at least one correct solution in 'k' attempts.
- executability checks in CODESYS V3.5.

The replication validated the results. Each model generated code for all 21 tasks across three independent runs. Code Llama produced syntactically valid outputs but struggled with executability. Platypus2 and StableCode usually failed, they generated C-like code or non-executable code. Table 6.1.1 presents the replication results of open-source code-LLMs.

Table 6.1.1: Tran's Replication Experiment Results of Code LLMs

Model	**Problems**	**ST Code Present**	**Compiled**	**Not compiled**	**No ST Code Present**
Code LLama	63	60	15	45	3
Playtypus2	63	26	0	26	37
StableCoder	63	3	0	3	60

Table 6.1.2: Tran's Experiment Pass@k Scores for $k = 1, 2, 3$ without Prompt Optimization

Model	**Pass@1 (%)**	**Pass@2 (%)**	**Pass@3 (%)**
Code LLama	25%	44%	58.53%
Playtypus2	0%	0%	0%
StableCoder	0%	0%	0%

Table 6.1.3: Tran's Experiment Pass@k Scores for $k = 1, 2, 3$ with Prompt Optimization

Model	**Pass@1 (%)**	**Pass@2 (%)**	**Pass@3 (%)**
Code LLama	4.76%	9.37%	13.83%
Playtypus2	0%	0%	0%
StableCoder	0%	0%	0%

Code LLMs like StableCode and Platypus2, which are mostly trained on general programming languages, failed to generalize PLC-specific ST syntax. These results reinforce that while universal LLMs can partially generate ST code, their performance remains inconsistent.

6.1.2 Experiment 2: Few-Shot Learning without Prompt Optimization

Based on the background discussed in section 6.1.1, another experiment is conducted to test whether few-shot learning can be used to enhance ST code generation. However, in this experiment, five representative few-shot examples of ST code are used. These are typical examples of PLC program constructs including timers (TON), IF-ELSE branching, arithmetic operations, and basic state-based logic. This experiment also conducted on the same representative 21 benchmark tasks (three times repetition) for evaluating 3 open source code LLMs (Code Llama, Platypus2, and StableCode). Evaluation was performed through executability testing in CODESYS V3.5 and by computing pass@k metrics and CodeBERTScore.

Table 6.1.4: Comparison of Different Code LLMs on ST Code Tasks with Few-shot Learning

Model	**Problems**	**ST Code Present**	**Compiled**	**Not Compiled**	**No ST Code Present**
Code LLama	63	63	27	36	0
Platypus2	63	46	11	35	17
StableCoder	63	34	3	31	29

As shown in Table 6.1.4, few-shot prompting outperformed the zero-shot baseline, which was not optimised for prompting. The syntax correctness also improved as shown in Table 6.1.5, and the build success ratio increased for Code Llama, Platypus2, and StableCode as shown in Table 6.1.6. The results of pass@k (where as k =1,2,3) are demonstrated in Table 6.1.7. For CodeBERTScore analysis, the median of the F3 score from the three runs was selected. Subsequently, the average of the 21 medians was calculated as shown in Table 6.1.8.

Table 6.1.5: Summary of Syntax Correctness Improvement Across Experiments (Experiment1 and Experiment 2)

Model Name	**Tran's Experiment**	**Our Experiment**	**Improvement**
Code LLama	23.81	42.86	+19.05%
Playtypus	0	17.46	+17.46%
StableCoder	0	4.76	+4.76 %

On the whole, this experiment shows that few-shot prompting can be used as an effective alternative to manual prompt optimization and that it allows models to more effectively generalize ST syntax and common PLC patterns

Table 6.1.6: Build Success Comparison Across Experiments (Experiment1 and Experiment 2)

Model Name	**Tran's Experiment**	**Our Experiment**	**Improvement**
Code LLama	15	27	+19.05%
Playtypus2	0	11	+17.46%
StableCoder	0	3	+4.76%

Table 6.1.7: Few-Shot Learning Pass@k Scores for $k = 1, 2, 3$ without Prompt Optimization

Model	**Pass@1 (%)**	**Pass@2 (%)**	**Pass@3 (%)**
Code LLama	45%	70.12%	84.06%
Playtypus2	18.33%	33.56%	46.16%
StableCoder	5.0%	9.83%	14.49%

Table 6.1.8: The average of the medians of Precision, Recall, F1, and F3 from the three runs without Prompt optimization

Model	**Precision**	**Recall**	**F1**	**F3**
Code Llama	0.7930	0.7840	0.7870	0.7850
Platypus2	0.7946	0.7687	0.7804	0.7709
StableCode	0.7038	0.7442	0.7220	0.7395

through in-context learning. However, the model-wise improvements are not uniform as they exibit some of the common errors (e.g., misuse of RETURN, improper termination of control structure).

6.1.3 Experiment 3: Few-Shot Learning with Prompt Optimization

In this experiment, the main objective is to compare performance of code LLMs when combining few-shot learning with prompt optimization to generate ST code. It is aimed to determine how the example based context and optimized prompts can enhance the quality and accuracy of generated ST code. Table 6.1.9 demonstrates the results of Code LLama, Platypus2, and StableCoder.

For CodeBERTScore analysis, the median of the F3 score from the three runs is selected. Table 6.1.10 shows the average of the three runs of the Precision, Recall, F1, and F3 over all the three models, which are reported as the mean of the median of each value. These measures represent the

Table 6.1.9: Comparison of Different Code LLMs on ST Code Tasks with Prompt Optimization and Few-shot Learning

Model	Problems	ST Code Present	Compiled	Not Compiled	No ST Code Present
Code LLama	63	60	35	25	03
Platypus2	63	56	18	38	07
StableCoder	63	40	00	40	23

Table 6.1.10: Average of Medians of Precision, Recall, F1, and F3 from Three Runs with Prompt Optimization

Model	Precision	Recall	F1	F3
Code Llama	0.8144	0.8405	0.8259	0.8373
Platypus2	0.8215	0.8352	0.8274	0.8335
StableCode	0.7525	0.7545	0.7521	0.7538

Table 6.1.11: Few-Shot Learning Pass@k Scores for $k = 1, 2, 3$ with Prompt Optimization

Model	Pass@1 (%)	Pass@2 (%)	Pass@3 (%)
Code LLama	58.33%	83.05%	93.28%
Playtypus2	30%	51.36%	66.45%
StableCoder	0%	0%	0%

Table 6.1.12: Build Success Comparison Across Experiments (Experiment 2 and Experiment 3)

Model	Experiment 2	Experiment 3	Improvement
Code LLama	27	35	+12.70%
Platypus2	11	18	+11.11%
StableCoder	3	0	–4.76%

quality and rightness of the produced ST code, considering both syntactic and semantic accurateness. These findings confirm that prompt optimization increases the overall quality of the generated code in all models. StableCoder, is still behind in terms of overall code quality. The Pass@k scores point at the significant improvement of Code LLama and Platypus2 under the prompt optimization as shown in Table 6.1.11. Overall, Experiment 6.1.3 shows that few-shot learning with prompt optimization significantly improved the quality of code generated by the model. In contrast, Tran's experiment found that using prompt optimization decreased their model's performance (Table 6.1.3)

6.1.4 Experiment 4: Fine-Tuning of Open-Source Code LLMs on ST code for Code Completion

This experiment explores the effect of fine-tuning on code LLMs for ST code completion using a *fill-in-the-middle* (FIM) approach. In contrast to the earlier experiments 6.1.1- 6.1.3 that focused on zero-shot and few-shot learning both with and without prompt optimization, this experiment investigates the possibility of domain-specific adaptation through fine-tuning whether it can improve both syntactic and functional correctness of generated ST code.

Dataset Preparation

In the experiment, the hybrid dataset was created by combing a publicly available set of ST programs with a smaller chunk of synthetic ST code generated via pipeline described in Chapter 4. This approach was a reasonable choice because, while the pipeline enables large-scale dataset generation, the full synthetic corpus was not accessible at the moment. The hybrid dataset was then a reasonable choice, as it could be fine-tuned, and synthetic augmentation input could also be tried as well.

The publicly available dataset was in form of open repositories of ST code samples typically used in automation education and open community projects as discussed in section 4.2.4. It is consisting of simple control structures such as counters, timers, arithmetic operations, and simple state machines. These were real programming examples, but small and sparse, especially in more complex industrial applications.

A smaller synthetic dataset chunk was generated with the pipeline described in Chapter 4 in order to complement the publicly available data. The TwinCAT parser was used to validate all of the synthetic programs before being included to make sure they are syntactically correct.

Data Preprocessing

The publicly available ST code sources (OSCAT, OpenPLC, PLC-BEAD) are typically encapsulated in XML format such as .tcPOU files. These files contains both the ST program code and metadata, such as program name, category, description, author, and creation date. These XML encapsulated files are suitable for storage and sharing, but cannot be used directly for model fine-tuning, which requires plain .st source code files. To overcome this limitation, a custom tool was created which parsed the XML encapsulated ST code files into plain .ST files.

Manually extracting ST programs from XML files are very time-consuming,

and datasets can contain hundreds of programs with numerous nested tags. It is also prone to error, as any single malformed or missing XML escape character can easily result in the breakdown of the ST syntax, and may even be unreliable, as all datasets need not have the same XML schema. To address these challenges, a custom Python preprocessing tool was developed, also representing another small key contribution of this work. This custom tool provides a bit of important functionality that explicitly defeats the challenge of processing XML-encapsulated ST programs. The tool has the following features;

First, it is automated (no manual copying and formatting of code, which is time-consuming and can include errors). It extracts clean .st files from .tcPOU or XML data and does not necessarily need manual copying and formatting of code.

Second, it is syntactically correct during the extraction process, i.e., the resulting .st files can be reliably compiled and parsable in TwinCAT, this is critical in industrial applications and in model optimization.

Third, the tool is batch processable and thus hundreds of programs can be converted at once, massively boosting efficiency and scalability.

Finally, it provides comprehensive logging and error-handling, documenting failed extractions or inconsistencies and allowing users to track and analyze them seamlessly.

Collectively, these features make the tool a potent and essential component of the data preparation pipeline, and allows reproducible and high-quality data to be utilized in subsequent experiment. It is also written in Python and can be easily run at the command line, making it a versatile and scalable system to prepare datasets. It uses XML data parsing and navigates the XML structure with the help of Python's xml.etree.ElementTree module. The `<POU>` component is found to scrap out the name of the program and this element is used to build the output .st file. The tool is then used to create valuable content, processing all of the XML elements and code in `<Declaration>` and `<ST>` tags and non-essential metadata, including author information or comments. To be syntactically correct, this text is written to output directory with a suffix 'st' that does not strip out the formatting of the code with suitable indenting and line breaks. Finally, it has a powerful error handling mechanism, which is used to capture any error during parsing or writing and show an informative message to allow the user to trace any issue and correct any error which may occur.

The Algorithm 1 outlines the process of taking in XML input and producing the output .st.

Algorithm 1 Convert .tcPOU or XML to Plain .st File

Require: XML file containing an ST program
Ensure: Plain `.st` file
1: Load XML file using `ET.parse()`
2: Locate the `<POU>` element to extract the program name
3: Initialize an empty list `content` to store extracted text
4: **for** each element in the XML tree **do**
5: **if** element is `<Declaration>` or `<ST>` **then**
6: Extract text content and append to `content` list
7: **end if**
8: **end for**
9: Construct output file path using program name
10: Write all collected content to the `.st` output file
11: Handle any parsing or writing errors

Hybrid Dataset Composition

After preprocessing steps, the plain '.st' files were combined with a synthetic ST code subset generated using the pipeline descibed in Chapter 4. This formed a hybrid dataset for fine-tuning. Public dataset is preprocessed programs from OSCAT, OpenPLC, and PLC-BEAD while synthetic subset programs are generated via the pipeline. The hybrid dataset thus contains approximately 2000 ST programing files.

Methodology and Results Analysis

The dataset consisted of raw ST code files without associated prompts. To make it suitable for fine-tuning, each code file was split into three segments:

- **Prefix (`<PRE>`)**: The initial part of the code that provides context.
- **Middle (`<MID>`)**: The segment to be predicted by the model during training. This portion is masked and serves as the target for the model.
- **Suffix (`<SUF>`)**: The ending portion of the code, providing additional context for predicting the middle segment.

The models are trained to predict the middle section given the prefix and suffix context. Special tokens `<PRE>`, `<MID>`, and `<SUF>` were added to the tokenizer to support this structure. During training, *fill-in-the-middle* transformation is applied probabilistically to each example using the special permutation mode (SPM), as implemented in `fim.permute()`. This allows

the model to learn to generate the middle section of code given its surrounding context, improving its ability to perform code completion in industrial ST programming tasks.

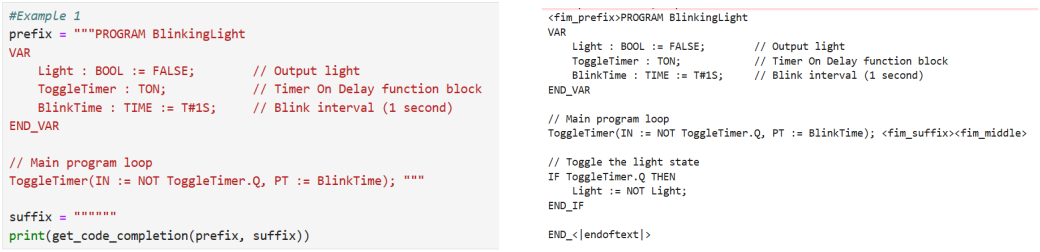
Initially, two models are fine-tuned on the prepared dataset for demonstratation:

- Code Llama (7B Instruct)
- Phi-3.5

Table 6.1.13 described the Hyperparameter settings for this experiment.

Table 6.1.13: Hyperparameters Used for Fine-Tuning in Experiment 6.1.4

Hyperparameters	**Value**
Sequence Length	2048 tokens
Max Steps	1000
Batch Size	1
Gradient Accumulation Steps	2
Learning Rate	5e-5
LR Scheduler	Cosine
Weight Decay	0.01
Warmup Steps	30
Evaluation Frequency	Every 200 steps
Checkpoint Saving Frequency	Every 200 steps
Logging Frequency	Every 200 steps
Mixed Precision	BF16 enabled, FP16 disabled
FIM Rate	0.5
FIM SPM Rate	0.5
Flash Attention	Disabled



```
#Example 1
prefix = """PROGRAM BlinkingLight
VAR
    Light : BOOL := FALSE;        // Output light
    ToggleTimer : TON;            // Timer On Delay function block
    BlinkTime : TIME := T#1S;     // Blink interval (1 second)
END_VAR

// Main program loop
ToggleTimer(IN := NOT ToggleTimer.Q, PT := BlinkTime); """

suffix = """"""
print(get_code_completion(prefix, suffix))
```

(a) Input prefix/suffix (Blinking Light)

```
<fim_prefix>PROGRAM BlinkingLight
VAR
    Light : BOOL := FALSE;        // Output light
    ToggleTimer : TON;            // Timer On Delay function block
    BlinkTime : TIME := T#1S;     // Blink interval (1 second)
END_VAR

// Main program loop
ToggleTimer(IN := NOT ToggleTimer.Q, PT := BlinkTime); <fim_suffix><fim_middle>

// Toggle the light state
IF ToggleTimer.Q THEN
    Light := NOT Light;
END_IF

END_<|endoftext|>
```

(b) Generated output

Figure 6.1.1: ST-Phi3.5 FIM experiment result for the Blinking Light program. The model successfully completed the missing middle section, producing functional logic.

```
prefix = """PROGRAM TrafficLight
VAR
    // Outputs
    RedLight    : BOOL := FALSE;
    YellowLight : BOOL := FALSE;
    GreenLight  : BOOL := FALSE;

    // State handling
    State       : INT := 0;       // 0=Red, 1=Green, 2=Yellow
    Timer       : TON;            // Timer block
END_VAR
CASE State OF
    0: // Red"""

suffix = """"""
print(get_code_completion(prefix, suffix))
```

(a) Input prefix/suffix (Motor Control)

```
<fim_prefix>PROGRAM TrafficLight
VAR
    // Outputs
    RedLight    : BOOL := FALSE;
    YellowLight : BOOL := FALSE;
    GreenLight  : BOOL := FALSE;

    // State handling
    State       : INT := 0;       // 0=Red, 1=Green, 2=Yellow
    Timer       : TON;            // Timer block
END_VAR
CASE State OF
    0: // Red<fim_suffix><fim_middle>
        RedLight := TRUE;
        GreenLight := FALSE;
        YellowLight := FALSE;
        Timer(IN := TRUE, PT := T#10S); // Timer to transition to Green
        IF Timer.Q THEN
            State := 1; // Transition to Green
        END_IF
```

(b) Generated output

Figure 6.1.2: ST-Phi3.5-FIM experiment result for the Trafficlight program. The model successfully completed the missing middle section, producing functional logic.

```
#Example 1
prefix = """PROGRAM BlinkingLight
VAR
    Light : BOOL := FALSE;        // Output light
    ToggleTimer : TON;            // Timer On Delay function block
    BlinkTime : TIME := T#1S;     // Blink interval (1 second)
END_VAR

// Main program loop
ToggleTimer(IN := NOT ToggleTimer.Q, PT := BlinkTime); """

suffix = """"""
print(get_code_completion(prefix, suffix))
```

(a) Input prefix/suffix (Blinking Light)

```
<s> <fim_prefix>PROGRAM BlinkingLight
VAR
    Light : BOOL := FALSE;        // Output light
    ToggleTimer : TON;            // Timer On Delay function block
    BlinkTime : TIME := T#1S;     // Blink interval (1 second)
END_VAR

// Main program loop
ToggleTimer(IN := NOT ToggleTimer.Q, PT := BlinkTime); <fim_suffix><fim_middle>

IF ToggleTimer.Q THEN
    Light := NOT Light;
END_IF; <fim_suffix>

// EOF <fim_suffix></s>
```

(b) Generated output

Figure 6.1.3: ST-llama-FIM experiment result for the Blinking Light program. The model successfully completed the missing middle section, producing functional logic.

```
prefix = """PROGRAM TrafficLight
VAR
    // Outputs
    RedLight    : BOOL := FALSE;
    YellowLight : BOOL := FALSE;
    GreenLight  : BOOL := FALSE;

    // State handling
    State       : INT := 0;       // 0=Red, 1=Green, 2=Yellow
    Timer       : TON;            // Timer block
END_VAR
CASE State OF
    0: // Red"""

suffix = """"""
print(get_code_completion(prefix, suffix))
```

(a) Input prefix/suffix (Motor Control)

```
<s> <fim_prefix>PROGRAM TrafficLight
VAR
    // Outputs
    RedLight    : BOOL := FALSE;
    YellowLight : BOOL := FALSE;
    GreenLight  : BOOL := FALSE;

    // State handling
    State       : INT := 0;       // 0=Red, 1=Green, 2=Yellow
    Timer       : TON;            // Timer block
END_VAR
CASE State OF
    0: // Red<fim_suffix><fim_middle>
        // Transition to green after 3 seconds
        Timer(IN := TRUE, PT := T#3S);
        IF Timer.Q THEN
            State := 1;
            Timer(IN := FALSE);
        END_IF
```

(b) Generated output

Figure 6.1.4: ST-llama-FIM experiment result for the Trafficlight program. The model successfully completed the missing middle section, producing functional logic.

To illustrate the capabilities of the *fill-in-the-middle* (FIM) model, we initially applied five candidate exmples for code completion on both fine-tuned models to check their ability on code completion tasks. Each example consisted of an input prefix containing variable declarations and partial program code, alongside the corresponding model-generated output filling in the missing logic. For visual representation of the results on both models we took 2 examples of each model's output as shown in Figs. 6.1.1, 6.1.2,6.1.3, and 6.1.4, respectively. By comparing the input and output side by side, it is evident from the initial results demonstrate that the fine-tuned models are able to generate missing part of the ST code.

6.2 Discussion

This section provides an in-depth discussion of the results obtained from experiments mentioned in section 6.1, focusing on the comparative analysis of code-generating LLMs, namely Code LLaMA, Platypus2, and StableCoder, in the context of Structured Text programming. The experiments gradually delved into zero-shot learning, few-shot learning, prompt-optimized few-shot learning, and fine-tuning for code completion. It is argued that the code generation quantity, syntactic correctness, semantic correctness, and success probabilities have been improved and gives the insight of model-specific behaviors.

6.2.1 ST Code Generation and Compilation Performance

Experiment 6.1.1 evaluated the open source code LLMs as per Tran's et al. [109] findings, in which no example context was presented. Code LLama produced ST code for 60 problems, of which 15 out of 60 compiled successfully and 45 out of 60 failed to compile even with a syntactically correct code. Platypus2 produced ST code for 26 problems, of which 0 were compiled without errors, and StableCoder produced ST code for 3 problems, of which 0 compiled successfully. These findings suggest that there are significant differences in the initial syntactic abilities of the models, which represents the effect of pre-training data and structure on zero-shot ST code generation.

Experiment 6.1.2 evaluated few-shot learning without prompt optimization, which presented the models with a limited set of few-shots examples. The compilation success of Code LLama went up to 27 from 15, Platypus2 from 0 to 11, and StableCoder went up slightly from 0 to 3 (as non-compilable ST code was still the predominating one). This proofs that even limited example based context can enhance the production of syntactically correct ST code especially the model with robust base capabilities. The fact that the number of ST code in the models has increased is also an indication that the few-shot examples help models in learning task patterns and generate more reliable code.

Experiment 6.1.3 used prompt optimization and few-shot learning together. Code LLama produced 60 ST codes, of which 35 were compiled successfully. Platypus2 produced 56 ST codes, 18 of which compiled successfully and StableCoder produced 40 ST codes, none of which compiled successfully. The comparison of Experiment 6.1.2 and Experiment 6.1.3 showed that prompt optimization enhanced the probability of code execution, particularly with the stronger models like codellama. The findings emphasize the fact that prompt optimization has the most significant effect on the quality of code and the success of model execution of models already exhibiting good zero-shot or few-shot performance.

6.2.2 Syntax and Semantic Correctness

The syntax correctness, quantified by the proportion of ST codes compiled without errors is clearly on an upward trend in the three experiments. For Code LLama syntax correctness was improved in experiment 6.1.2 upto +19.05% from Tran's experiment and in experiment 6.1.3 it has improved upto +12.70% from experiment 6.1.2. Platypus2 improved from 0% to +17.46% in experiment 6.1.2, and in experiment6.1.3 improved upto +11.11%. The

StableCoder did not change much because of the weaknesses in architecture and training data coverage. These findings show that few-shot learning cannot produce a significant increase of syntax correctness in models with the ability to achieve great gains, less powerful models need more strategies to realize significant gains.

The semantic correctness was measured such as Precision, Recall, F1, and F3. In Experiment 6.1.1, Code LLama scored moderate in semantics (Precision 0.793, F1 0.787), compared to Platypus2 and StableCoder which were also very small or close to zero. In Experiment 6.1.2, with few-shot learning, these metrics were improved by all the models, the F1 score of Code LLama improved to 0.825, and the Platypus2 score improved to 0.780. In Experiment 6.1.3, prompt optimization improved the semantic correctness, and the F1 of Code LLama and Platypus2 were 0.826 and 0.827. StableCoder slightly improved (F1 = 0.752) but did not yet achieve the level of practical use. In general, the findings indicate that few shot learning and prompt optimization are critical towards the generation of semantically correct ST code, and more powerful models are more advantageous.

6.2.3 Pass@k Analysis

Pass@k scores provide insight into the likelihood that at least one of the top k generated solutions is correct. In experiment 6.1.1, Code LLama achieved Pass@3 = 58.53%, while Platypus2 and StableCoder were essentially ineffective. Few-shot learning without prompt optimization (Experiment 6.1.2) improved Pass@3 for Code LLama to 84.06% and for Platypus2 to 46.16%, indicating that in-context fewshot examples substantially increase the probability of correct solutions among the top candidates. StableCoder's improvement was minimal (Pass@3 = 14.49%), highlighting architectural limitations. In Experiment 6.1.3, combining few-shot learning with prompt optimization resulted in the highest Pass@k scores. The Code LLama achieved Pass@3 = 93.28%, Platypus2 reached 66.45%, while StableCoder remained at 0%. These results emphasize that prompt optimization, in addition to few-shot examples, maximizes the likelihood of producing correct solutions, particularly for models with strong base performance. Models with weaker foundational capabilities, such as StableCoder, show limited gains, suggesting that prompt optimization alone cannot overcome inherent model limitations.

6.2.4 Comparative Analysis Across Experiments

Several key observations emerge from the comparison of the above-mentioned experiments. Across all evaluated models, each successive experimental con-

figuration resulted in incremental performance gains, with improvements observed in syntax correctness, semantic correctness, and Pass@k scores when moving from few-shot learning to few-shot learning with prompt optimization. These gains indicate that progressively richer contextual guidance consistently enhances code generation performance.

A consistent hierarchy among the models was maintained throughout the experiments. Code LLaMA outperformed all other models across every evaluation metric, followed by Platypus2 and StableCoder. Although few-shot learning improved results for all models, the relative performance differences remained evident and were further amplified by prompt optimization, particularly for the stronger models.

Few-shot learning proved especially beneficial for weaker models, enabling them to generate correct code in cases where zero-shot prompting failed. In contrast, the impact of prompt optimization was more selective: while it significantly improved Pass@k and semantic correctness for strong models, it had limited effect on models with poor base capabilities, suggesting that prompt engineering alone cannot fully compensate for fundamental model limitations.

Finally, consistent patterns were observed in ST code generation across experiments. Strong models, particularly Code LLaMA, maintained a high ratio of generated code relative to the total number of problems and demonstrated better compilation success. Weaker models, on the other hand, generated fewer ST programs and struggled more frequently with compilation, highlighting the dependence of reliable code generation on model capacity.

Experiment 6.1.4 explored the fine-tuning of large language models, including Code Llama and and Phi-3.5, on a domain-specific ST code dataset. Unlike the previous experiments, which focused on zero-shot and few-shot generation, this experiment adapted the models to the task of code completion using a *fill-in-the-middle* (FIM) approach. Due to the nature of the dataset, metrics such as Pass@k and few-shot improvements were not applicable. Preliminary results indicate that fine-tuning substantially improved code completion tasks.

These expermental results demonstrated that domain-specific fine-tuning can significantly enhance ST code generation capabilities. Few-shot learning and prompt optimization improve performance incrementally, fine-tuning allows models to better adapt to industrially relevant code completion tasks, providing a practical path to improve code generation reliability.

Chapter 7

Conclusion

This study investigated the potential of Large Language Models for generating ST code in industrial automation contexts. The research combined theoretical analysis, empirical experiments, and practical implementations to examine the feasibility, limitations, and industrial applicability of modern code-generating LLMs. The core contributions of this thesis are as follows:

The first contribution is the development of industrial feasibility of *PEFTpyCODER*, a Parameter-Efficient Fine-Tuned model designed as an in-house code assistant for industrial settings. This contribution built directly upon the feasibility study, which demonstrated that code LLMs can be adapted locally without relying on Cloud-based infrastructures. The study employed a combination of Parameter-Efficient Fine-Tuning, Low-Rank Adaptation, and 4-bit quantization techniques to fine-tune an open-source model (Phi-3-mini-128k-instruct, 3.8B parameters). The goal was to evaluate whether SOTA models could achieved superior performance using only commodity hardware while ensuring no data leave the firms premises. The resulting PEFTpyCODER model achieved superior performance on two widely accepted python benchmarks (HumanEval and MBPP) compared to 5 times larger models, which confirmed that lightweight fine-tuning delivered high accuracy with drastically lower computational cost. The proposed approach demonstrated the feasibility of deploying AI-assisted code generation tools locally within an organization, eliminating the need to transmit proprietary data to Cloud or external servers. This contribution thus validated the technical and organizational viability of developing domain-adapted LLMs for industrial firms, laying the foundation for future adaptation of similar approach to the ST programming language used in PLCs systems.

The second contribution of this research was the design and implementation of a synthetic Structured Text code generation pipeline that enabled the automated creation and validation of a large, high-quality dataset for

training and evaluation of code LLMs. The pipeline employed Azure OpenAI GPT-4 instance to generate TwinCAT-compliant ST projects through natural language prompts and validated the output using TwinCAT compiler and a iterative feedback loop. When compilation errors occured, the system automatically extracted and reformulated the error messages, generated ST code, and exacted error lines from the code to guide the model for correction using a iterative feedback loop. The final validation stage applied TcUnit test cases to ensure semantic accuracy. This approach resulted a dataset which produced syntactically and functionally correct code, thereby established a domain-specific ST code generation mechanism. The pipeline's novelty lied in its integration of LLM generation with compiler and testing tools, made it the first of its kind tailored for industrial Structured Text programming.

The third contribution comprised a comprehensive experimental evaluation of the proposed approaches, with a focus on adapting LLMs to ST code. The experiments replicated and extended previous research on domain-specific fine-tuning. The study investigated few-shot learning with and without prompt engineering, as well as the fine-tuning of two open-source code LLMs for code completion tasks using both synthetic and publicly available ST datasets. The initial results demonstrated that effective prompt engineering and PEFT-based fine-tuning substantially improved model performance, produced higher-quality, syntactically valid, and contextually coherent code outputs. These experiments provided empirical evidence that LLMs could be successfully adapted to specialized industrial programming languages using limited computational resources.

Another contribution of this research focused on licensing and innovation management aspects. This work examined the legal, ethical, and strategic dimensions of deploying Large Language Models in industrial contexts, highlighting the critical importance of licensing, IP, and innovation management for the industrial adoption of code LLMs.

These findings demonstrate that, with appropriate model selection, prompt engineering, and task-specific fine-tuning, LLMs can generate ST code that is syntactically and semantically meaningful for industrial automation tasks. This study contributes to a deeper understanding of model capabilities, task-specific learning strategies, and practical considerations for deployment, thereby establishing a foundation for integrating LLMs into industrial settings.

7.1 Future Research Directions

Building upon this work, future research can explore:

- Extending evaluation to additional industrial programming languages and large-scale codebases to validate the generalizability of findings. IEC 61131-3 has other languages like Ladder Diagram (LD) and Function Block Diagram (FBD). Structured Text is usually considered the most amenable to LLMs due to its textual nature. However, one could imagine generating other representations or even translating ST to Ladder diagram descriptions. Combining multi-modal learning (text and graphics) would be challenging but potentially very useful (since many PLC engineers still use ladder logic heavily).

- Investigating advanced fine-tuning strategies, such as reinforcement learning with human feedback (RLHF), to enhance semantic correctness and task-specific performance.

- Integrating natural language context and documentation into training data to improve comprehension and maintainability of generated code.

- Developing automated and scalable evaluation pipelines for fine-tuned models in industrial automation contexts.

- Initially, the goal of the synthetic ST dataset creation pipeline was to generate a large-scale dataset comprising approximately 200,000 samples of ST code. However, due to computational and practical constraints, this target could not be fully achieved within the current scope. The future work also plan to extend the dataset to this intended scale.

Bibliography

[1] Marah Abdin et al. "Phi-3 technical report: A highly capable language model locally on your phone". In: *arXiv preprint arXiv:2404.14219* (2024).

[2] Marah Abdin et al. "Phi-4 technical report". In: *arXiv preprint arXiv:2412.08905* (2024).

[3] Yonatan Gizachew Achamyeleh et al. "Bridging the PLC Binary Analysis Gap: A Cross-Compiler Dataset and Neural Framework for Industrial Control Systems". In: *arXiv preprint arXiv:2502.19725* (2025).

[4] Josh Achiam et al. "Gpt-4 technical report". In: *arXiv preprint arXiv:2303.08774* (2023).

[5] Josh Achiam et al. "Gpt-4 technical report". In: *arXiv preprint arXiv:2303.08774* (2023).

[6] Sandhini Agarwal et al. "gpt-oss-120b & gpt-oss-20b model card". In: *arXiv preprint arXiv:2508.10925* (2025).

[7] Wasi Uddin Ahmad et al. "Unified pre-training for program understanding and generation". In: *arXiv preprint arXiv:2103.06333* (2021).

[8] Loubna Ben Allal et al. "Santacoder: don't reach for the stars!" In: *arXiv preprint arXiv:2301.03988* (2023).

[9] Anthropic. *Claude*. 2023. URL: `https://claude.ai/`.

[10] Mikel Artetxe et al. "Efficient large scale language modeling with mixtures of experts". In: *arXiv preprint arXiv:2112.10684* (2021).

[11] Jacob Austin et al. "Program synthesis with large language models". In: *arXiv preprint arXiv:2108.07732* (2021).

[12] Mohammad Bavarian et al. "Efficient training of language models to fill in the middle". In: *arXiv preprint arXiv:2207.14255* (2022).

[13] Beckhoff Automation. *TwinCAT Chat*. 2024. URL: `https://www.beckhoff.com/en-en/products/automation/twincat-projects-with-ai-supported-engineering/`.

[14] Iz Beltagy, Kyle Lo, and Arman Cohan. "SciBERT: A pretrained language model for scientific text". In: *arXiv preprint arXiv:1903.10676* (2019).

[15] Loubna Ben Allal et al. *A framework for the evaluation of code generation models*. `https://github.com/bigcode-project/bigcode-evaluation-harness`. 2022.

[16] Prasad Birmole et al. "Designing and implementation of chemical mixing and filling bottles using PLC". In: *2018 Second International Conference on Inventive Communication and Computational Technologies (ICICCT)*. IEEE. 2018, pp. 436–439.

[17] Sid Black et al. "Gpt-neox-20b: An open-source autoregressive language model". In: *arXiv preprint arXiv:2204.06745* (2022).

[18] William Bolton. *Programmable logic controllers*. Newnes, 2015.

[19] Malte Brettel et al. "How virtualization, decentralization and network building change the manufacturing landscape: an industry 4.0 perspective." In: *FormaMente* 12 (2017).

[20] Tom Brown et al. "Language models are few-shot learners". In: *Advances in neural information processing systems* 33 (2020), pp. 1877–1901.

[21] Ilias Chalkidis et al. "LEGAL-BERT: The muppets straight out of law school". In: *arXiv preprint arXiv:2010.02559* (2020).

[22] Shubham Chandel et al. "Training and evaluating a jupyter notebook data science assistant". In: *arXiv preprint arXiv:2201.12901* (2022).

[23] Mark Chen et al. "Evaluating large language models trained on code". In: *arXiv preprint arXiv:2107.03374* (2021).

[24] Henry Chesbrough. "Open innovation: a new paradigm for understanding industrial innovation". In: *Open innovation: Researching a new paradigm* 400 (2006), pp. 0–19.

[25] Aakanksha Chowdhery et al. "Palm: Scaling language modeling with pathways". In: *Journal of Machine Learning Research* 24.240 (2023), pp. 1–113.

[26] Fenia Christopoulou et al. "PanGu-Coder: Program Synthesis with Function-Level Language Modeling". In: *CoRR* (2022).

[27] Colin Clement et al. "PyMT5: multi-mode translation of natural language and Python code with transformers". In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 2020, pp. 9052–9065.

[28] Cognition AI. *Devin*. 2024. URL: https://www.cognition.ai/blog/introducing-devin.

[29] Gheorghe Comanici et al. "Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities". In: *arXiv preprint arXiv:2507.06261* (2025).

[30] International Electrotechnical Commission et al. "IEC 61131-3: 2013 programmable controllers-Part 3: programming languages". In: *Geneva, Switzerland* (2013).

[31] Programmable Controllers—Part. "Programmable Controllers—Part 3: Programming Languages". In: *Standard IEC* (2013), pp. 61131–3.

[32] James Curzon et al. "Privacy and Artificial Intelligence". In: 2.2 (2021), pp. 96–108. DOI: 10.1109/TAI.2021.3088084.

[33] Begum Karaci Deniz et al. "Unleashing developer productivity with generative AI". In: *McKinsey Digital* 7 (2023).

[34] Jacob Devlin et al. "Bert: Pre-training of deep bidirectional transformers for language understanding". In: *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*. 2019, pp. 4171–4186.

[35] Peng Di et al. "Codefuse-13b: A pretrained multi-lingual code large language model". In: *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*. 2024, pp. 418–429.

[36] Mohamad Fakih 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*. 2024, pp. 192–203.

[37] Zhangyin Feng et al. "Codebert: A pre-trained model for programming and natural languages". In: *arXiv preprint arXiv:2002.08155* (2020).

[38] Daniel Fried et al. "Incoder: A generative model for code infilling and synthesis". In: *arXiv preprint arXiv:2204.05999* (2022).

[39] GitHub. *Copilot*. 2021. URL: https://github.com/features/copilot.

[40] Keith Goffin and Rick Mitchell. *Innovation management*. Bloomsbury Publishing, 2025.

[41] Google. *Gemini*. 2023. URL: https://gemini.google.com/.

[42] Aaron Grattafiori et al. “The llama 3 herd of models”. In: *arXiv preprint arXiv:2407.21783* (2024).

[43] Suriya Gunasekar et al. “Textbooks are all you need”. In: *arXiv preprint arXiv:2306.11644* (2023).

[44] Daya Guo et al. “Graphcodebert: Pre-training code representations with data flow”. In: *arXiv preprint arXiv:2009.08366* (2020).

[45] Daya Guo et al. “Unixcoder: Unified cross-modal pre-training for code representation”. In: *arXiv preprint arXiv:2203.03850* (2022).

[46] Aaron Haag et al. “Training LLMs for Generating IEC 61131-3 Structured Text with Online Feedback”. In: *2025 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code)*. IEEE. 2025, pp. 65–71.

[47] Vivek Hajarnavis and Ken Young. “An investigation into programmable logic controller software design techniques in the automotive industry”. In: *Assembly Automation* 28.1 (2008), pp. 43–54.

[48] Bing Han et al. “Domain-specific translation tool from structured text to C source code with code readability enhancement in programmable logic controllers”. In: *Concurrency and Computation: Practice and Experience* 36.15 (2024), e8100.

[49] Zeyu Han et al. “Parameter-Efficient Fine-Tuning for Large Models: A Comprehensive Survey”. In: *Transactions on Machine Learning Research* 2024 (2024).

[50] Zhenyu He et al. “Let the code llm edit itself when you edit the code”. In: *arXiv preprint arXiv:2407.03157* (2024).

[51] Dan Hendrycks et al. “Measuring Coding Challenge Competence With APPS”. In: *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*.

[52] Edward J Hu et al. “Lora: Low-rank adaptation of large language models.” In: *ICLR* 1.2 (2022), p. 3.

[53] Siming Huang et al. “OpenCoder: The Open Cookbook for Top-Tier Code Large Language Models”. In: *CoRR* (2024).

[54] Binyuan Hui et al. “Qwen2. 5-coder technical report”. In: *arXiv preprint arXiv:2409.12186* (2024).

[55] Aaron Hurst et al. “Gpt-4o system card”. In: *arXiv preprint arXiv:2410.21276* (2024).

[56] Hamel Husain et al. “Codesearchnet challenge: Evaluating the state of semantic code search”. In: *arXiv preprint arXiv:1909.09436* (2019).

[57] Maliheh Izadi et al. “Language models for code completion: A practical evaluation”. In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 2024, pp. 1–13.

[58] Siyuan Jiang et al. “aiXcoder-7B: A Lightweight and Effective Large Language Model for Code Processing”. In: *2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE. 2025, pp. 215–226.

[59] Arpan Kumar Kar, P. S. Varsha, and Shivakami Rajan. “Unravelling the Impact of Generative Artificial Intelligence (GAI) in Industrial Applications: A Review of Scientific and Grey Literature”. In: *Global Journal of Flexible Systems Management* 24.4 (2023), pp. 659–689. ISSN: 0974-0198. DOI: 10.1007/s40171-023-00356-x. URL: https://doi.org/10.1007/s40171-023-00356-x.

[60] 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)*. IEEE. 2023, pp. 1–8.

[61] Alexander Lavin et al. “Technology readiness levels for machine learning systems”. In: *Nature Communications* 13.1 (2022), p. 6039.

[62] Hung Le et al. “Coderl: Mastering code generation through pretrained models and deep reinforcement learning”. In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 21314–21328.

[63] Jay Lee, Behrad Bagheri, and Hung-An Kao. “A cyber-physical systems architecture for industry 4.0-based manufacturing systems”. In: *Manufacturing letters* 3 (2015), pp. 18–23.

[64] Jinhyuk Lee et al. “BioBERT: a pre-trained biomedical language representation model for biomedical text mining”. In: *Bioinformatics* 36.4 (2020), pp. 1234–1240.

[65] Patrick Lewis et al. “Retrieval-augmented generation for knowledge-intensive nlp tasks”. In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 9459–9474.

[66] Raymond Li et al. “Starcoder: may the source be with you!” In: *arXiv preprint arXiv:2305.06161* (2023).

[67] Yuanzhi Li et al. “Textbooks are all you need ii: phi-1.5 technical report”. In: *arXiv preprint arXiv:2309.05463* (2023).

[68] Yujia Li et al. “Competition-level code generation with alphacode”. In: *Science* 378.6624 (2022), pp. 1092–1097.

[69] GNU General Public License. “Gnu general public license”. In: *Retrieved December* 25 (1989), p. 2014.

[70] Zihan Liu et al. “Agents4PLC: Automating Closed-loop PLC Code Generation and Verification in Industrial Control Systems using LLM-based Agents”. In: *arXiv preprint arXiv:2410.14209* (2024).

[71] Anton Lozhkov et al. “Starcoder 2 and the stack v2: The next generation”. In: *arXiv preprint arXiv:2402.19173* (2024).

[72] Shuai Lu et al. “Codexglue: A machine learning benchmark dataset for code understanding and generation”. In: *arXiv preprint arXiv:2102.04664* (2021).

[73] Qinyu Luo et al. “Repoagent: An llm-powered open-source framework for repository-level code documentation generation”. In: *arXiv preprint arXiv:2402.16667* (2024).

[74] Ziyang Luo et al. “WizardCoder: Empowering Code Large Language Models with Evol-Instruct”. In: *ICLR*. 2024.

[75] Hinin Yu Lwin and U Hla Myo Htay. “Design and Simulation of Automated Packaging Machine Process Control by Using PLC”. In: *Int. J. Trend Sci. Res. Dev* 3 (2019), pp. 1423–1426.

[76] Scott Morgan. “The Effect of Artificial Intelligence Code Generation on Software Developer Productivity”. PhD thesis. Marymount University, 2025.

[77] Dimitris Mourtzis, Ekaterini Vlachou, and NJPC Milas. “Industrial big data as a result of IoT adoption in manufacturing”. In: *Procedia cirp* 55 (2016), pp. 290–295.

[78] Fiona Fui-Hoon Nah et al. “Generative AI and ChatGPT: Applications, challenges, and AI-human collaboration”. In: *Journal of Information Technology Case and Application Research* 25.3 (2023), pp. 277–304. DOI: 10.1080/15228053.2023.2233814. eprint: https://doi.org/10.1080/15228053.2023.2233814. URL: https://doi.org/10.1080/15228053.2023.2233814.

[79] Tung Thanh Nguyen et al. “A statistical semantic language model for source code”. In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 2013, pp. 532–542.

[80] Erik Nijkamp et al. “Codegen: An open large language model for code with multi-turn program synthesis”. In: *arXiv preprint arXiv:2203.13474* (2022).

[81] Erik Nijkamp et al. “Codegen2: Lessons for training llms on programming and natural languages”. In: *arXiv preprint arXiv:2305.02309* (2023).

[82] OpenAI. *ChatGPT*. 2024. URL: `https://chatgpt.com/`.

[83] Long Ouyang et al. “Training language models to follow instructions with human feedback”. In: *Advances in neural information processing systems* 35 (2022), pp. 27730–27744.

[84] Gnana Swathika OV et al. “Critical review of SCADA and PLC in smart buildings and energy sector”. In: *Energy reports* 12 (2024), pp. 1518–1530.

[85] Ruchika Pandey et al. “Transforming software development: Evaluating the efficiency and challenges of github copilot in real-world projects”. In: *arXiv preprint arXiv:2406.17910* (2024).

[86] Sida Peng et al. “The impact of ai on developer productivity: Evidence from github copilot”. In: *arXiv preprint arXiv:2302.06590* (2023).

[87] Alec Radford et al. “Language models are unsupervised multitask learners”. In: *OpenAI blog* 1.8 (2019), p. 9.

[88] Veselin Raychev, Martin Vechev, and Eran Yahav. “Code completion with statistical language models”. In: *Proceedings of the 35th ACM SIGPLAN conference on programming language design and implementation*. 2014, pp. 419–428.

[89] Eric Reis. “The lean startup”. In: *New York: Crown Business* 27 (2011), pp. 2016–2020.

[90] Adnan Riaz et al. “Industrial Feasibility of PEFTpyCoder: A Parameter-Efficient Fine-Tuned Model for In-House Code Assistance”. In: *2025 International Joint Conference on Neural Networks (IJCNN)*. IEEE. 2025, pp. 1–10.

[91] Tal Ridnik, Dedy Kredo, and Itamar Friedman. “Code Generation with AlphaCodium: From Prompt Engineering to Flow Engineering”. In: *arXiv e-prints* (2024), arXiv–2401.

[92] Everett M Rogers, Arvind Singhal, and Margaret M Quinlan. “Diffusion of innovations”. In: *An integrated approach to communication theory and research*. Routledge, 2014, pp. 432–448.

[93] Baptiste Roziere et al. “Code llama: Open foundation models for code”. In: *arXiv preprint arXiv:2308.12950* (2023).

[94] Jerome H Saltzer. “The origin of the “MIT license””. In: *IEEE Annals of the History of Computing* 42.4 (2020), pp. 94–98.

[95] ByteDance Seed et al. “Seed-coder: Let the code model curate data for itself”. In: *arXiv preprint arXiv:2506.03524* (2025).

[96] Meet Shah, Rajat Shenoy, and Radha Shankarmani. “Natural language to python source code using transformers”. In: *2021 International Conference on Intelligent Technologies (CONIT)*. IEEE. 2021, pp. 1–4.

[97] Bo Shen et al. “Pangu-coder2: Boosting large language models for code with ranking feedback”. In: *arXiv preprint arXiv:2307.14936* (2023).

[98] Andrew Sinclair. “License profile: Apache license, version 2.0”. In: *IFOSS L. Rev.* 2 (2010), p. 107.

[99] Kuldeep Singh, Sheshadri Chatterjee, and Marcello Mariani. “Applications of generative AI and future organizational performance: The mediating role of explorative and exploitative innovation and the moderating role of ethical dilemmas and environmental dynamism”. In: *Technovation* 133 (2024), p. 103021. ISSN: 0166-4972. DOI: https://doi.org/10.1016/j.technovation.2024.103021. URL: https://www.sciencedirect.com/science/article/pii/S0166497224000713.

[100] Mário de Sousa and Adriano Carvalho. “An IEC 61131-3 compiler for the MatPLC”. In: *EFTA 2003. 2003 IEEE Conference on Emerging Technologies and Factory Automation. Proceedings (Cat. No. 03TH8696)*. Vol. 1. IEEE. 2003, pp. 485–490.

[101] Sergey M Staroletov. “Grammar-based testing a process-oriented extension of the IEC 61131-3 structured text language”. In: *2022 International Russian Automation Conference (RusAutoCon)*. IEEE. 2022, pp. 863–869.

[102] Jack Stilgoe and David Guston. “Responsible research and innovation”. In: MIT Press, 2016.

[103] Weisong Sun et al. “Source code summarization in the era of large language models”. In: *arXiv preprint arXiv:2407.07959* (2024).

[104] Alexey Svyatkovskiy et al. “Intellicode compose: Code generation using transformer”. In: *Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*. 2020, pp. 1433–1443.

[105] CodeGemma Team et al. “CodeGemma: Open Code Models Based on Gemma”. In: *arXiv e-prints* (2024), arXiv–2406.

[106] Hassaan Th H Thabet and Maysara A Qasim. “Proposed industrial concept for automating a food production process using PLC”. In: *2013 International Conference on Electrical Communication, Computer, Power, and Control Engineering (ICECCPCE)*. IEEE. 2013, pp. 147–151.

[107] Romal Thoppilan et al. “LaMDA: Language Models for Dialog Applications”. In: *arXiv preprint arXiv:2201.08239* (2022).

[108] Hugo Touvron et al. “Llama: Open and efficient foundation language models”. In: *arXiv preprint arXiv:2302.13971* (2023).

[109] Kilian Tran et al. “Generating PLC Code with Universal Large Language Models”. In: *2024 IEEE 29th International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE. 2024, pp. 1–8.

[110] Ashok Urlana et al. *No Size Fits All: The Perils and Pitfalls of Leveraging LLMs Vary with Company Size*. 2024. arXiv: 2408.01444 [cs.CY]. URL: https://arxiv.org/abs/2408.01444.

[111] Ashish Vaswani et al. “Attention is all you need”. In: *Advances in neural information processing systems* 30 (2017).

[112] Yue Wang et al. “Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation”. In: *arXiv preprint arXiv:2109.00859* (2021).

[113] Yue Wang et al. “CodeT5+: Open Code Large Language Models for Code Understanding and Generation”. In: *arXiv preprint* (2023).

[114] Yue Wang et al. “Codet5+: Open code large language models for code understanding and generation”. In: *arXiv preprint arXiv:2305.07922* (2023).

[115] Yue Wang et al. “CodeT5Mix: A Pretrained Mixture of Encoder-decoder Transformers for Code Understanding and Generation”. In: (2022).

[116] BigScience Workshop et al. “Bloom: A 176b-parameter open-access multilingual language model”. In: *arXiv preprint arXiv:2211.05100* (2022).

[117] Yuchen Xia et al. “Control industrial automation system with large language models”. In: *arXiv preprint arXiv:2409.18009* (2024).

[118] Frank F Xu et al. “A systematic evaluation of large language models of code”. In: *Proceedings of the 6th ACM SIGPLAN international symposium on machine programming*. 2022, pp. 1–10.

[119] Si Xu et al. “HETHUB: A Distributed Training System with Heterogeneous Cluster for Large-Scale Models”. In: *arXiv preprint arXiv:2405.16256* (2024).

[120] An Yang et al. “Qwen3 technical report”. In: *arXiv preprint arXiv:2505.09388* (2025).

[121] Donghao Yang et al. “A Multi-Agent Framework for Extensible Structured Text Generation in PLCs”. In: *arXiv preprint arXiv:2412.02410* (2024).

[122] Zhongzhi Yu et al. “Edge-llm: Enabling efficient large language model adaptation on edge devices via unified compression and adaptive layer voting”. In: *Proceedings of the 61st ACM/IEEE Design Automation Conference*. 2024, pp. 1–6.

[123] Daoguang Zan et al. “CERT: Continual Pre-Training on Sketches for Library-Oriented Code Generation”. In: *arXiv e-prints* (2022), arXiv–2206.

[124] Sebastian Zhao et al. “Codinggenie: A proactive llm-powered programming assistant”. In: *arXiv preprint arXiv:2503.14724* (2025).

[125] Qinkai Zheng et al. “Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x”. In: *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 2023, pp. 5673–5684.

[126] Yue Zheng et al. “A review on edge large language models: Design, execution, and applications”. In: *ACM Computing Surveys* 57.8 (2025), pp. 1–35.

[127] Albert Ziegler et al. “Measuring github copilot’s impact on productivity”. In: *Communications of the ACM* 67.3 (2024), pp. 54–63.