# Evaluating Large Language Models for Functional and Maintainable Code in Industrial Settings: A Case Study at ASML

Yash Mundhra
*Delft University of Technology*
Delft, Netherlands
y.mundhra@student.tudelft.nl

Max Valk
*ASML*
Veldhoven, Netherlands
max.valk@asml.com

Maliheh Izadi
*Delft University of Technology*
Delft, Netherlands
m.izadi@tudelft.nl

***Abstract*—Large language models have shown impressive performance in various domains, including code generation across diverse open-source domains. However, their applicability in proprietary industrial settings, where domain-specific constraints and code interdependencies are prevalent, remains largely unexplored. We present a case study conducted in collaboration with the leveling department at ASML to investigate the performance of LLMs in generating functional, maintainable code within a closed, highly specialized software environment.**

**We developed an evaluation framework tailored to ASML's proprietary codebase and introduced a new benchmark. Additionally, we proposed a new evaluation metric, *build@k*, to assess whether LLM-generated code successfully compiles and integrates within real industrial repositories. We investigate various prompting techniques, compare the performance of generic and code-specific LLMs, and examine the impact of model size on code generation capabilities, using both match-based and execution-based metrics. The findings reveal that prompting techniques and model size have a significant impact on output quality, with few-shot and chain-of-thought prompting yielding the highest build success rates. The difference in performance between the code-specific LLMs and generic LLMs was less pronounced and varied substantially across different model families.**

***Index Terms*—Code Generation, Large Language Models, LLM, Prompting Techniques, Few-shot, Chain-of-Thought, Evaluation**

## I. Introduction

The rise of large language models (LLMs) has significantly influenced software engineering practices, with tools like GitHub Copilot and ChatGPT already supporting developers in writing, refactoring, and understanding code [1]. Despite their promising capabilities, most LLMs are trained on public, open-source data and evaluated on standardized benchmarks that focus on generating straightforward, standalone code snippets [2], [3]. As a result, their effectiveness in proprietary, domain-specific industrial settings remains unclear. In practice, industrial software systems often involve legacy components, tightly coupled modules, specialized APIs, and unique naming conventions [4], [5]. All of which pose significant challenges for LLM-based code generation.

In this work, we investigate whether and how LLMs can be effectively applied in real-world industrial code bases that lie outside their training dataset. Specifically, we explore the applicability of LLM-based code generation in the ASML leveling department, where code operates within a complex, layered architecture. We examine whether LLMs can generate interdependent, buildable, and functional code in a proprietary repository with no or limited prior exposure to its domain-specific terminology or structure.

Existing literature has largely focused on function-level or benchmark tasks with clear input-output mappings [6]. Few studies have explored repository-level code generation in closed industrial environments [7], [8]. This gap leaves open questions around the feasibility of LLMs for code generation in an industrial setting. Aspects such as prompt engineering, model specialization, model size, and functional correctness have been studied very minimally in repository-level settings.

We address these open questions through an in-depth empirical study conducted at ASML. By evaluating LLMs across prompting strategies and model configurations, and by introducing a novel evaluation metric (build@k), we provide practical insights into the challenges and opportunities of adopting LLMs for domain-specific software generation.

Our contributions are as follows.

- Our study demonstrates how LLMs perform when tasked with generating code in a closed, domain-specific repository within a proprietary domain.
- We created a custom benchmark using ASML's internal code to evaluate the generation capabilities of LLMs.
- We propose a novel metric, *build@k*, to evaluate whether generated code successfully compiles and builds, providing a more realistic measure of usefulness than traditional similarity-based metrics.
- We compare zero-shot, few-shot, and chain-of-thought (CoT) prompting techniques and find that the latter two significantly outperform zero-shot in this domain.
- We show that code-specific LLMs generally outperform generic ones, though the gap varies across model families.
- Lastly, our research highlights that larger models tend to perform better, but gains diminish beyond a certain size (around 14B parameters), suggesting diminishing returns on scaling.

## II. Background and Related Work

The pivotal research by Hindle et al. [9] proved that software, although theoretically complex, is indeed predictable through statistical modeling. It is therefore not surprising that code completion and code generation have become some of the most thoroughly researched applications of LLMs in software engineering, leveraging this predictable nature to generate effective code recommendations [10], [11]. Additionally, a survey conducted by Hou et al. [1] emphasized that current research predominantly focuses on applying LLMs during the software development phase of the engineering life cycle.

Within this growing field of research, a distinction can be made between *general-purpose* LLMs and *code-specific* LLMs. General-purpose LLMs, such as GPT-4 [12] and LLaMa 3 [13], are trained on a vast and diverse corpus of text, including web data, code, documents, and news articles, which provides them with a broad knowledge base. These models have demonstrated impressive performance in various software engineering tasks, including code writing, understanding, and reasoning. In contrast, code-specific LLMs are typically trained on a massive corpus of programming data or fine-tuned from a general-purpose LLM using a large amount of programming data [2]. By focusing on programming-related tasks and challenges, these code-specific models have achieved even better performance than generic LLMs when it comes to generating functionally correct code.

Since the release of CodeX, a decoder-only language model fine-tuned for programming tasks, [14], research on LLMs for code generation has accelerated. The introduction of the HumanEval benchmark, created to evaluate functional correctness from docstrings, played a key role in this surge, becoming a standard for assessing model performance in code synthesis. In 2024, Zhao et al. [15] released CodeGemma, a decoder-only model built on Google's Gemma architecture, trained on 500 billion tokens of primarily code data, achieving state-of-the-art performance in code generation and completion tasks.

That same year, the DeepSeek-AI team launched DeepSeek-Coder-V2 [16], an open-source model pre-trained on 6 trillion additional tokens, enhancing its coding and mathematical reasoning capabilities while maintaining performance in general language tasks. The model outperformed all open-source counterparts and matched leading closed-source models like GPT-4 Turbo. Shortly after, Hui et al. [17] released Qwen2.5-Coder, a decoder-only model from Alibaba, which significantly improved upon its predecessor by being pre-trained on over 5.5 trillion tokens of code-centric data.

While LLMs show strong capabilities in code generation, benchmarks like HumanEval focus on simple tasks, such as generating standalone functions or statements. Software development, however, involves complex dependencies and interdependent code units [18]. Jimenez et al. [19] assessed LLMs in realistic settings, where models resolved issues in GitHub repositories, with the best model, Claude 2, resolving only 1.96% of issues. This highlights the need for improved domain-specific code generation, a field still largely underexplored.

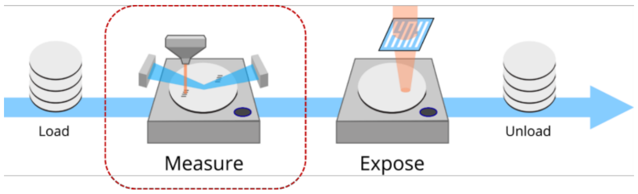

Fig. 1. Simplified view of the life of a wafer in the scanner. The wafer is loaded, measured, exposed, and then unloaded [23].

To tackle more challenging coding scenarios, Du et al. [18] introduced ClassEval, which involves generating classes based on descriptions, test suites, and benchmark solutions, using a class skeleton as a blueprint. Despite its contributions, ClassEval treats classes as isolated units, relying only on common libraries likely included in LLM training data. Yu et al. [20] proposed CoderEval, evaluating LLMs on pragmatic code generation using non-standalone functions from real-world projects. Yet, CoderEval is limited to line- and function-level tasks. Deshpande et al. [21] introduced RepoClassBench, assessing LLMs on generating non-standalone classes within a repository's context, offering a more realistic reflection of real-world scenarios. Nevertheless, despite these advances, research on industrial-scale code generation remains relatively limited and underexplored.

When using LLMs, the quality of the prompt plays a crucial role in shaping the output they generate [22]. In software engineering tasks, three popular prompting techniques are used the most: zero-shot, few-shot, and chain-of-thought prompting [1]. Zero-shot prompting asks the model to tackle a task without any examples, relying on its existing knowledge to figure things out. Few-shot prompting provides a few examples in the prompt to help the model understand the task better, leading to more accurate results. Chain-of-thought prompting takes it a step further by encouraging the model to break down complex tasks into logical steps, which is great for handling intricate problems that require sequential thinking. While these techniques have proven effective in various scenarios, their impact on generating repository-level domain-specific code is not yet well-studied.

## III. Problem and Industrial Context

Our study is conducted at the ASML leveling department. ASML is a global leader in photolithography systems used for semiconductor manufacturing. A critical component of the lithography process is metrology, which involves measuring the wafer with extreme high precision (See Figure 1). Within the metrology cluster, the leveling department plays an important role by measuring the vertical position of the wafer using level sensors. This precise measurement is utilized to keep the wafer in focus during the exposure stage, thereby ensuring optimal accuracy and precision in the manufacturing process.

Within its leveling department, software plays a critical role, allowing them to process large amounts of data collected by the leveling sensors and perform computations on them. The

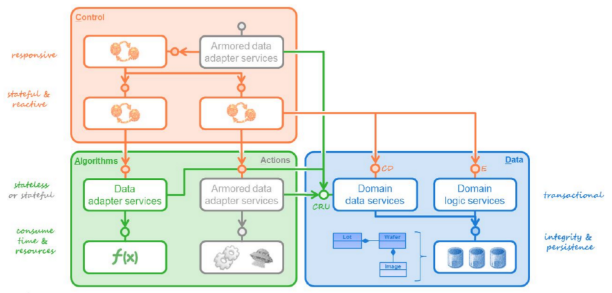

Fig. 2. DCA architecture used in the ASML leveling software separating the data, control, and algorithm components [24]

software in the department is built using the Data Control and Algorithms (DCA) architecture that divides the software into three distinct components: data, control, and algorithms, ensuring that the code base is easily maintainable and has a clear separation of concerns.

The data component is responsible for data persistence, storage, and management. It involves storing and retrieving data related to the wafer and scanner, as well as managing its lifecycle, including creation, update, and deletion. The control component is responsible for decision-making, control logic, and task management. It involves planning, scheduling, and coordinating tasks and subtasks, as well as managing the flow of data and algorithms. It also ensures that the ordering constraints of the system are met. Finally, the algorithm component is responsible for data transformations, calculations, and processing. It involves executing mathematical operations, solving optimization problems, and performing other complex data processing tasks. An overview of the DCA architecture is presented in Figure 2.

ASML's need for an automated coding tool stems from the repetitive and time-consuming task of writing glue code, which connects various components of the DCA architecture. While this task is relatively straightforward, it can be quite time-consuming for developers, pulling them away from more creative and complex problem-solving activities. Automating these routine coding processes would significantly boost productivity, allowing developers to focus on higher-value work and improving overall efficiency. However, the challenge lies in the specialized terminology and strict interface contracts, which make it difficult for LLMs to effectively handle this type of code.

In this study, we address a key automation challenge: generating new "*garage*" interfaces within the data component of the DCA architecture that handle storing and retrieving data from a repository. When the control or algorithm components need specific pieces of data, they access it through these garages. There are three main types: store-garages for storage, retrieve-garages for retrieval, and store-retrieve garages that handle both operations. Although the implementation of a garage follows a fairly standard pattern, it still demands significant manual effort and domain expertise. This makes it an ideal candidate for automation using LLMs, potentially streamlining the process and reducing the workload on developers.

The current coding process presents several challenges that hinder both efficiency and scalability. One issue is the low level of automation: although many garages share similar structures, there is no existing tooling to support their automatic generation based on prior examples. Additionally, code interdependencies further complicate development. Creating a new garage often requires integrating with multiple other components, demanding cross-repository knowledge and careful coordination to avoid breaking builds.

## A. *Opportunity for Automation*

These challenges make the garage generation task a promising candidate for automation through LLMs. The task exhibits repeated structural patterns that models can learn and reproduce, while its success criteria are clearly defined: the generated garage must compile, pass all tests, and meet ASML's code quality standards. Moreover, the task reflects broader industrial challenges, where proprietary architectures and domain-specific knowledge play a central role.

## B. *Research Questions*

To explore the potential of LLMs in automating garage generation, we focus on three key research questions:

- ***RQ1:* To what extent do different prompting techniques affect the performance of generated domain-specific code?**
  This question investigates the extent to which various prompting techniques influence the quality and correctness of code generated for proprietary environments. By systematically varying the prompting approach, this research question seeks to identify optimal strategies for generating high-quality and functional code from LLMs.
- ***RQ2:* How do generic LLMs compare to code-specific LLMs in generating domain-specific, interdependent code?**
  This question examines the relative performance of generic LLMs, such as the Gemma models trained on diverse datasets, versus code-specific models like CodeGemma, which are optimized for software engineering tasks. The evaluation focuses on their ability to generate buildable, functionally correct, and high-quality code.
- ***RQ3:* To what extent does the size of the large language model influence the performance of generating domain-specific code?**
  Model size is often correlated with improved performance in natural language processing tasks. This question examines whether larger models, with their increased capacity for learning complex patterns, offer tangible benefits for generating domain-specific code. By systematically comparing models of varying sizes, we aim to identify the trade-offs between computational cost and generation quality. Gaining insight into this relationship is crucial for

selecting models that balance efficiency and performance in industrial settings.

## IV. Methodology

To evaluate the feasibility of using LLMs for domain-specific code generation at ASML, we designed a practical framework for assessing model performance in real industrial settings. The goal of our approach is to generate functional "garage" interfaces in ASML's leveling codebase using LLMs. To support the generation and evaluation of such garages, we follow a multi-step approach:

1) **Benchmark Dataset Creation**: Extraction of 156 garage examples from ASML's proprietary codebase, with corresponding context files and test cases.
2) **Experimental Setup**: Controlled experiments using multiple prompting strategies, model types, and sizes.
3) **Evaluation Strategy**: Quantitative and qualitative assessments using match-based, execution-based, and human-evaluated metrics.
4) **Result Analysis**: Statistical aggregation and interpretation of performance outcomes.

### A. Benchmark Dataset Creation

We start off by constructing a benchmark dataset consisting of 156 garages from the leveling repository. Each garage's file path and implementation were stored in the benchmark. Additionally, unit tests were collected to evaluate functional correctness; however, only 42 garages (approximately 27%) had associated tests.

To provide contextual information for each garage, we recursively collected all files referenced through import statements, assigning a depth value based on their distance in the import hierarchy. Files directly imported by the garage were given a depth of 1, and so on. To reduce noise, auto-generated files with a depth greater than 2 were excluded. This filtering ensured that only the most relevant human-written files were retained for the benchmark.

Given the limited context window of many LLMs, especially when dealing with long files (often exceeding 2000 lines), we summarized these files using the Qwen2.5-Coder-32B-Instruct model. Additionally, we computed embeddings for all context files using the BGE-M3 model and prioritized them using cosine similarity to the prompt embedding. Finally, token counts were calculated to determine which files could fit within the available context window.

The overall format of the benchmark dataset is shown below:

```
[
  {
    "name": "<Garage_Name>",
    "path": "<Garage_FilePath>",
    "component": "<Garage_Component>",
    "solution_code":
    ↪  "<Garage_Implementation>",
    "related_files": [
      {
        "file_path": "<CF_FilePath>",
        "depth": "<CF_Depth>",
        "implementation": "<CF_Impl>",
        "embedding": "<CF_Embedding>",
        "Qwen2.5-Tokens": "<CF_QwenTokens>",
        "DeepSeek-Tokens": "<CF_DSTokens>",
        "Gemma-Tokens": "<CF_GemmaTokens>"
      }, ..., {}
    ],
    "test_directory":
    ↪  "<Garage_TestDirectory>",
    "test_file_name": "<Garage_TestName>"
  }, ..., {}
]
```

The garages we collected for the benchmark dataset exhibit a wide range of sizes in terms of lines of code. Figure 3 presents a histogram illustrating the distribution of garage sizes and their corresponding frequencies. As evident from the figure, the majority of garages are small in size, containing less than 100 lines of code. Garages with more than 100 lines of code are less common, with only a small proportion of the benchmark falling into this category.

### B. Experimental Setup

The following section describes the experimental setup that we used to answer the three research questions, providing a clear and concise overview of the study's design and implementation.

*1) RQ1: Prompting Technique:* The first experiment examined the influence of prompting techniques on generated code quality. Five distinct prompting techniques were selected:

- Zero-shot prompting
- One-shot prompting
- Few-shot prompting
- One-shot chain-of-thought prompting
- Few-shot chain-of-thought prompting

Zero-shot prompting [25] involves instructing a language model to perform a task using only a task description, without

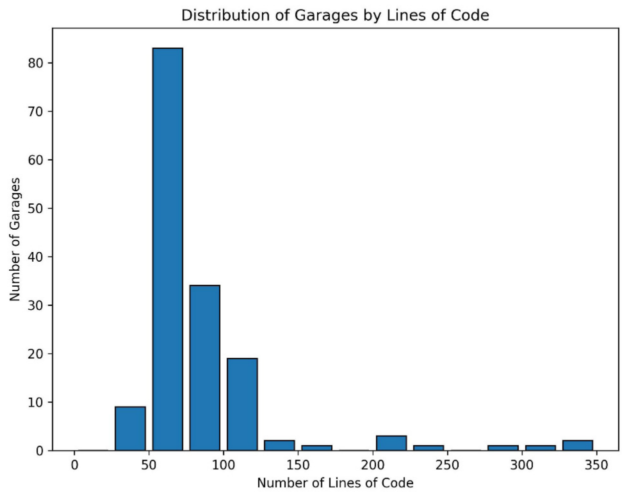

Fig. 3. Histogram with the distribution of garages in the benchmark by the number of lines of code.

providing any examples. The model relies entirely on its pre-trained knowledge to generate a response. In contrast, one-shot and few-shot prompting [26] enhance task understanding by including one or a few input-output examples, respectively. These examples serve as demonstrations to guide the model's behavior. While few-shot prompting can significantly improve performance on complex tasks, it also increases input length and is sensitive to the quality and selection of examples.

Chain-of-thought (CoT) prompting [27] builds on these techniques by encouraging the model to generate intermediate reasoning steps. In one-shot CoT prompting, a single example includes both the task and a step-by-step reasoning process, while few-shot CoT prompting provides multiple such examples. This structured approach helps larger models perform better on tasks requiring logical reasoning, such as arithmetic or commonsense inference.

All experiments for research question 1 were conducted using the Qwen2.5-Coder-32B-Instruct model with Q8_0 quantization and a 32k token context window, using default sampling parameters (temperature: 0.7, top-p: 0.8, top-k: 20). To manage the limited context window, we allocated 25k tokens for context files and 7k tokens for the prompt and output, based on the size of the largest prompt and garage in the dataset. Context files were prioritized first by import depth and then by cosine similarity to the prompt embedding, ensuring the most relevant files were included.

*2) RQ2: Generic versus Code-specific LLMs:* The second experiment compared the performance of generic LLMs and code-specific LLMs in generating garages within ASML. To ensure a fair comparison, we used code-specific models that were fine-tuned from the same generic base models, maintaining identical architectures. The models used for the experiments, as presented below, were evaluated in their default configurations using their standard sampling parameters with Q8_0 quantization.

- Qwen2.5-7B-Instruct (7B)
- Qwen2.5-Coder-7B-Instruct (7B)
- gemma-7b-it (7B)
- codegemma-7b-it (7B)
- DeepSeek-V2-Lite-Chat (16B)
- DeepSeek-Coder-V2-Lite-Instruct (16B)

All experiments used single-shot Chain-of-Thought (CoT) prompting due to its effectiveness in code generation [28]. Context file selection followed the same prioritization strategy as in the first experiment, with Qwen2.5 and DeepSeek models utilizing a 32k token context window, while Gemma models were limited to 8k tokens.

*3) RQ3: Model Size:* The third and final experiment that we conducted investigated the impact of large language model size variations on code generation performance within the ASML leveling department. To ensure a fair comparison, all models were selected from the same family, guaranteeing that the architecture remained consistent while the primary difference was the model size.

- Qwen2.5-Coder-0.5B-instruct
- Qwen2.5-Coder-1.5B-instruct
- Qwen2.5-Coder-3B-instruct
- Qwen2.5-Coder-7B-instruct
- Qwen2.5-Coder-14B-instruct
- Qwen2.5-Coder-32B-instruct

All models were configured uniformly using Q8_0 quantization, a 32k token context window, and default sampling parameters (temperature: 0.7, top-p: 0.8, top-k: 20), with 25k tokens allocated for context files and 7k for the prompt and output. Context file prioritization followed the same methodology as in the first two experiments, ensuring consistency across setups. Single-shot chain-of-thought prompting was used throughout, based on its demonstrated effectiveness in prior research [28].

## *C. Evaluation Strategy*

To assess the quality of generated code, we use a three-phase evaluation approach.

- **Match-based Metrics:** Assess the similarity between generated code and a reference implementation by comparing tokens, syntax structures, or exact outputs.
- **Execution-based Metrics:** Evaluate the functional correctness of the generated code by compiling and executing it, then comparing the observed behavior and outputs against expected results.
- **Manual Evaluation:** Involve human judgment to assess qualitative aspects such as readability, maintainability, and adherence to coding standards—factors not fully captured by automated similarity or execution-based metrics.

*1) Match-based Metrics:* The match-based metrics used in this study to compare the generated output and the reference implementation are BLEU, CodeBLEU, and ROUGE. The BLEU score [29] is calculated based on the number of matching n-grams between the generated code and the reference code. It then calculates the weighted mean of these matches to get the overall similarity score. CodeBLEU [30] enhances the traditional BLEU score by incorporating syntactic and semantic information specific to code. This is achieved by leveraging abstract syntax trees to represent code syntax and data-flow to capture code semantics. Finally, the ROUGE [31] score takes the recall value of n-gram overlaps between the generated code and the reference code. It provides a measure of how well the generated code matches the reference code in terms of content and structure.

*2) Execution-based Metrics:* Since code-similarity metrics offer limited insight into the functional correctness of the generated code, the second stage of the evaluation process involved assessing execution-based performance metrics. We use three main execution based metrics:

- **Buildability:** While traditional execution-based metrics, as noted by Chen et al. [32], typically focus on file- or function-level evaluations, but often neglect repository-level analysis. Consequently, evaluating the buildability of the generated code was deemed essential. To the best of our knowledge, there is no existing metric that assesses the buildability of generated code; therefore, we propose a novel performance metric, termed *build@k*, which draws inspiration from the pass@k metric [33]. The *build@k* metric takes $k$ code samples per garage. A garage is considered buildable if it successfully integrates into the project and passes the build pipelines; the total fraction of buildable garages is reported. Formula 1 as given below formally defines the *build@k* metric.

$$\text{build@k} = \frac{1}{N}\sum_{i=1}^{N}\begin{cases}1 & \text{if any of the } k \text{ generated code} \\ & \text{instances for garage}_i \text{ is built} \\ 0 & \text{otherwise}\end{cases} \tag{1}$$

- **Unit Tests:** To assess the functional correctness of the generated code, we execute the unit tests associated with the garage and employ the *pass@k* metric, as proposed by Kulal et al. [33]. The *pass@k* metric reports the total fraction of garages that are successfully solved (only consider garages with unit tests are considered for this metric). Formula 2 as given below formally defines the *pass@k* metric.

$$\text{pass@k} = \frac{1}{N}\sum_{i=1}^{N}\begin{cases} & \text{if any of the } k \text{ generated} \\ 1 & \text{code instances for garage}_i \\ & \text{passes all unit tests} \\ 0 & \text{otherwise}\end{cases} \tag{2}$$

- **Code Quality:** A significant gap in existing research is the evaluation of generated code in terms of code quality aspects, such as maintainability, readability, and performance efficiency [32]. We use the TICS tool, an evaluation platform used at ASML [34], to assess the quality of the generated code. The TICS tool (created by TIOBE) provides a comprehensive evaluation of code quality, enabling us to gauge the maintainability and readability of the generated code. This tool provides a more detailed assessment of the generated code's quality, moving beyond just functional correctness.
  The TICS tool assesses the generated code across a wide range of categories, including class interface, code organization, error handling, naming, and many others. Each rule is assigned a severity level, ranging from 1 to 10, with level 1 indicating the most critical issues (program errors) and level 10 representing the least critical rules (style issues). Issues assigned to levels 1-7 are deemed critical, whereas those assigned to levels 8-10 are considered non-critical [35]. Using the TICS score, we will measure two key metrics: the average number of violations per buildable garage and the average number of critical violations (levels 1–7) per buildable garage.

*3) Manual Evaluation:* As the final step in our evaluation strategy, manual evaluation enables us to assess aspects of generated code such as readability, maintainability, and adherence to coding standards and best practices, which are not fully captured by code-similarity and execution-based metrics.

TABLE I
MEAN BLEU AND CODEBLEU SCORES OF GENERATED CODE ACROSS FIVE DIFFERENT PROMPTING TECHNIQUES, WITH $k$ VALUES OF 1, 3, & 5.

| | **CodeBLEU** | | |
|---|---|---|---|
| | *k=1* | *k=3* | *k=5* |
| **Zero-Shot** | 0.320 | 0.360 | 0.380 |
| **One-Shot** | 0.395 | 0.440 | 0.456 |
| **Few-Shot** | 0.470 | 0.513 | 0.531 |
| **One-Shot CoT** | 0.456 | 0.502 | 0.512 |
| **Few-Shot CoT** | **0.483** | **0.523** | **0.534** |

## V. RESULTS

### A. RQ1: Prompting Techniques

*1) Match-based Metrics:* The CodeBLEU similarity results are presented in Table I, highlighting clear differences in performance across prompting techniques. Zero-shot and one-shot prompting yielded the lowest similarity scores across BLEU, CodeBLEU, and ROUGE metrics, indicating limited effectiveness in generating domain-specific code. In contrast, few-shot, one-shot CoT, and few-shot CoT prompting performed significantly better, with few-shot CoT leading. BLEU and ROUGE scores are not reported due to space limitations; however, they exhibited similar trends to the CodeBLEU scores.

Across all prompting methods, similarity scores improved as the number of generated outputs ($k$) increased, particularly from $k = 1$ to $k = 3$, suggesting that generating multiple outputs increases the likelihood of producing code closer to the reference. However, the gains between $k = 3$ and $k = 5$ were smaller, indicating diminishing returns with additional generations.

*2) Execution-based metrics:* In terms of the build@k metric, the results (as presented in table II) reveal a clear difference in the ability of the generated code to successfully integrate and compile within the project. Zero-shot prompting consistently performed the worst across all values of $k$, while few-shot prompting achieved the highest buildability at $k = 1$ and $k = 5$, generating 44 buildable garages at the highest setting. One-shot CoT prompting showed strong performance, particularly at $k = 3$, where it outperformed all other techniques. Overall, build@k scores improved with higher $k$ values across all prompting methods, indicating that generating multiple outputs increases the likelihood of producing buildable code. However, the rate of improvement varied, with the CoT-based prompting techniques showing smaller gains between $k = 3$ and $k = 5$.

The unit test results from Experiment 1 revealed that none of the generated garages passed the available tests, resulting in a pass@k score of 0.0 across all prompting techniques and values of $k$. This outcome was primarily due to two factors: a lack in the model's ability to generate garages that successfully build within the project, and a lack of unit test coverage among those that did. In most cases, the majority of buildable garages lacked associated unit tests, with coverage ranging from 0% to 21% depending on the prompting technique and $k$ value.

TABLE II

BUILD@K SCORE OF GENERATED CODE ACROSS FIVE DIFFERENT PROMPTING TECHNIQUES WITH $k$ VALUES OF 1, 3, AND 5.

| | build@k | | |
|---|---|---|---|
| | *k=1* | *k=3* | *k=5* |
| **Zero-Shot Prompting** | 0.01935 | 0.04516 | 0.08333 |
| **One-Shot Prompting** | 0.07189 | 0.14379 | 0.24837 |
| **Few-Shot Prompting** | **0.12820** | 0.21795 | **0.28205** |
| **One-Shot CoT Prompting** | 0.12179 | **0.23077** | 0.23718 |
| **Few-Shot CoT Prompting** | 0.10897 | 0.20513 | 0.21154 |

TABLE III

CODE QUALITY METRICS OF GENERATED CODE ACROSS FIVE DIFFERENT PROMPTING TECHNIQUES. **VPB** STANDS FOR VIOLATIONS PER BUILD.

| | | **Zero-Shot** | **One-Shot** | **Few-Shot** | **One-Shot CoT** | **Few-Shot CoT** |
|---|---|---|---|---|---|---|
| *k=1* | **VPB** | 2.00 | 1.27 | 1.05 | **1.00** | **1.00** |
| | **Critical VPB** | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| *k=3* | **VPB** | 2.29 | 1.32 | 1.15 | **1.06** | 1.13 |
| | **Critical VPB** | 0.143 | 0.05 | 0.03 | **0.03** | 0.06 |
| *k=5* | **VPB** | 2.00 | **1.11** | 1.14 | **1.11** | 1.12 |
| | **Critical VPB** | 0.00 | 0.00 | 0.05 | 0.05 | 0.06 |

Finally, in terms of code quality, the results presented in Table III show that zero-shot prompting consistently produced the lowest-quality code across all $k$ values. It had the highest violations per build, ranging from 2.00 to 2.29, and also introduced critical violations at $k = 3$ (0.143 per build), while all other techniques maintained lower rates. In contrast, one-shot CoT and few-shot CoT prompting achieved the best results, with violations per build close to or exactly 1.00 at $k = 1$, and minimal critical violations across all $k$ values. Overall, the results indicate that prompts containing examples improve the quality of the generated code.

## *B. RQ2: Code-Specific versus Generic LLMs*

*1) Match-based metrics:* The CodeBLEU scores (see figure 4) demonstrate that overall code-specific LLMs outperform their generic counterparts across all three model families; Qwen2.5, Gemma, and DeepSeek-V2. These code-specific models generate code that more closely matches the reference implementation. While the Qwen2.5 models showed minimal differences between the generic and code-specific variants, the Gemma and DeepSeek-V2 families exhibited substantial improvements with the code-specific fine-tuning. Notably, the DeepSeek-Coder-V2-Lite-Instruct model achieved a 105.8% increase in CodeBLEU score over its generic counterpart at $k = 1$. BLEU and ROUGE scores exhibited similar trends and have therefore been omitted for brevity.

*2) Execution-based metrics:* In terms of the build@k scores, the code-specific Qwen2.5 model slightly outperformed its generic counterpart across all $k$ values, though the differences were marginal. In contrast, the Gemma models exhibited a substantial gap in buildability: the code-specific variant generated significantly more buildable garages than the generic model, with a build@k score of 0.551 at $k = 5$ compared to just 0.019. Interestingly, the DeepSeek-V2 models showed the opposite pattern, where the generic model consistently achieved higher build@k scores than its code-specific counterpart. Despite the code-specific DeepSeek model producing more similar code, its lower buildability suggests that similarity alone is not a sufficient indicator of functional correctness, and that model suitability may vary depending on the evaluation criteria. Figure 5 presents the build@k scores from this experiment in a bar chart, providing a visual overview of the build performance of the generated code.

In terms of functional correctness, none of the generated garages successfully passed the unit tests, resulting in a score of 0 across all models and $k$ values. Unit test coverage was generally low across all model families, with only 28% of buildable garages from the Qwen2.5 models and 26% from the DeepSeek-V2 models having associated tests. The Gemma models showed the highest coverage at approximately 50%,

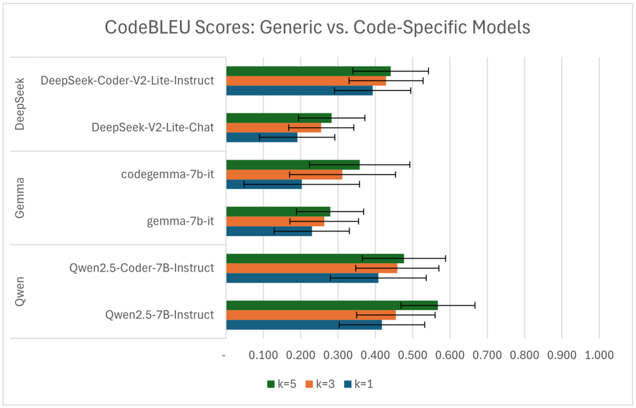

Fig. 4. CodeBLEU score of generated code using code-specific and generic LLMs, with k values of 1, 3, and 5

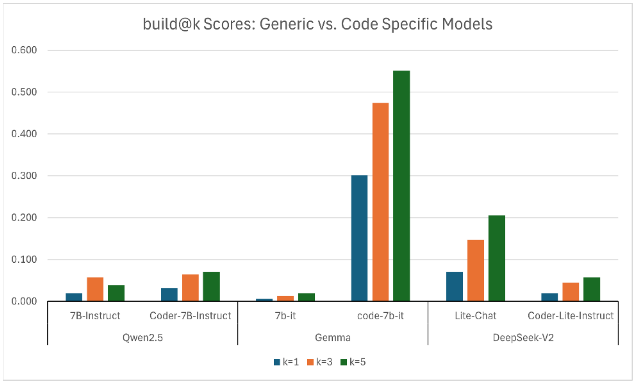

Fig. 5. Bar chart with build@k score of generated code comparing generic and code-specific LLMs for the Qwen2.5, Gemma, and DeepSeek-V2 families, with $k$ values of 1, 3, and 5.

yet still failed to produce any passing outputs. These results highlight a persistent gap in the functional correctness of generated code.

The code quality results from the second experiment reveal varying trends across the three model families. For the Qwen2.5 models, the code-specific variant consistently outperformed the generic model, maintaining a violations per build ratio close to 1.00 and incurring no critical violations across all $k$ values. In contrast, the Gemma models showed the opposite trend: the code-specific model exhibited significantly higher violations per build (exceeding 3.20) and a critical violations per build ratio above 0.55, while the generic model maintained lower values in both metrics. The DeepSeek-V2 models followed a similar pattern to Qwen2.5, with the code-specific model achieving lower violations per build (ranging from 1.43 to 2.00) and no critical violations, compared to the generic model, which consistently exceeded 3.00 violations per build and had a critical violation rate of approximately 0.5. These results suggest that while code-specific fine-tuning can improve code quality, its effectiveness varies depending on the model family.

### C. *RQ3: Model Size*

*1) Match-based metrics:* The CodeBLEU scores from experiment 3, as shown in Figure 6, indicate a positive correlation between model size and similarity. As the number of parameters increases, the generated code exhibits higher similarity to the reference solutions. The most significant improvements occur between the 0.5B and 3B models, with the rate of improvement reducing beyond the 14B model. The 32B model achieves the highest CodeBLEU scores across all $k$ values, although the performance gain over the 14B model is marginal, suggesting a plateauing effect at larger scales. Additionally, increasing the $k$ value from 1 to 3 yields noticeable improvements in similarity, while the difference between $k = 3$ and $k = 5$ is less substantial. These trends are consistent with those observed for BLEU and ROUGE, reinforcing the conclusion that larger models and moderate increases in $k$ enhance code generation quality, albeit with diminishing returns at the upper end of the model size spectrum.

*2) Execution-based metrics:* Figure 7 reveals that the build@k scores vary across different model sizes, indicating that buildability is not strictly correlated with model scale. Unlike the match-based metrics, which showed a consistent upward trend with increasing model size, the build@k scores fluctuate, with some smaller models outperforming larger ones. Notably, the 1.5B model achieved the highest build@k scores at $k = 3$ and $k = 5$, even surpassing the 14B and 32B models. The 0.5B model performed the worst, failing to produce any buildable code at $k = 1$ and only marginally improving at higher $k$ values. While the 14B and 32B models showed strong performance at $k = 1$, the 1.5B model managed to generate more buildable garages at higher $k$ values. Overall, increasing $k$ generally improved buildability across all models except the 0.5B variant, likely due to the increased chances of generating at least one buildable sample.

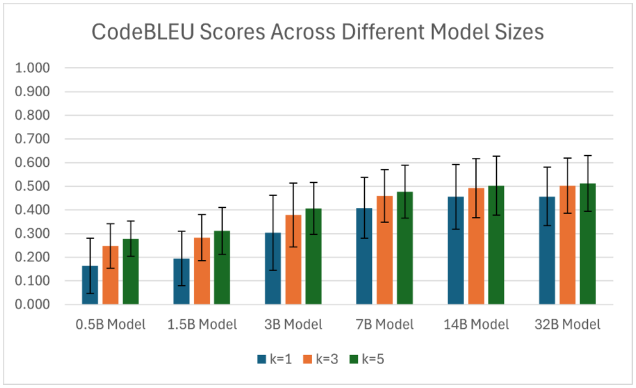

Fig. 6. Bar chart with mean CodeBLEU scores of generated code across different model sizes of Qwen2.5-Coder-Instruct, with $k$ values of 1, 3, & 5.

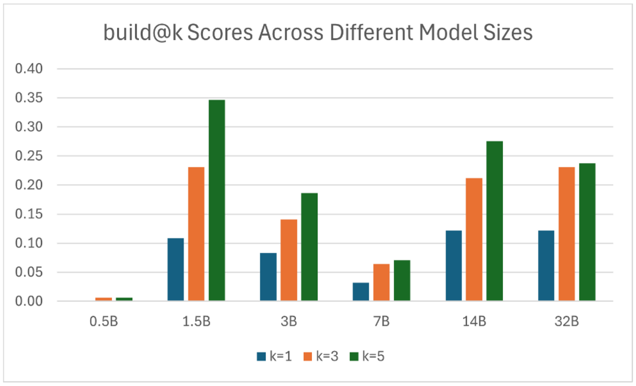

Fig. 7. Bar chart with build@k score of generated code across different model sizes of Qwen2.5-Coder-Instruct, with $k$ values of 1, 3, & 5.

Despite some garages being successfully built, none of the generated outputs passed the unit tests, indicating a lack of functional correctness. Since only 27% of the built garages had associated unit tests, it was not possible to draw firm conclusions about the models' ability to generate functionally correct code.

The final evaluation assesses the code quality of the generated outputs (Table V). The results show a clear distinction between smaller and larger models. The 0.5B, 1.5B, and 3B models consistently produce code with higher violation rates, exceeding a ratio of 2 violations per build across all $k$ values, and also exhibit elevated critical violation ratios, up to 1.0 in the case of the 0.5B model at $k = 5$. In contrast, the larger models (7B, 14B, and 32B) maintain a violations per build ratio close to 1.0 and demonstrate minimal critical violations, often registering 0. These findings suggest that larger models not only generate more syntactically correct code but also adhere more closely to coding standards and best practices, resulting in cleaner and safer outputs.

## VI. Discussion

The findings from the three experiments provide valuable insights into the effectiveness of prompting strategies, the com-

TABLE IV
CODE QUALITY METRICS OF GENERATED CODE, COMPARING CODE-SPECIFIC AND GENERIC LLMS WITH $k$ VALUES OF 1, 3, AND 5. **VPB** STANDS FOR VIOLATIONS PER BUILD.

| | | *k=1* | | *k=3* | | *k=5* | |
|---|---|---|---|---|---|---|---|
| | | **Generic** | **Code** | **Generic** | **Code** | **Generic** | **Code** |
| *Qwen2.5* | **VPB** | 1.67 | **1.00** | 1.67 | **1.00** | 1.67 | **1.09** |
| | **Critical VPB** | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| *Gemma* | **VPB** | **2.00** | 3.60 | **2.00** | 3.43 | **2.00** | 3.20 |
| | **Critical VPB** | 0.00 | 0.72 | 0.00 | 0.65 | 0.00 | 0.56 |
| *DeepSeek-V2* | **VPB** | 3.27 | **2.00** | 3.22 | **1.43** | 3.03 | **1.56** |
| | **Critical VPB** | 0.55 | 0.00 | 0.57 | 0.00 | 0.47 | 0.00 |

TABLE V
CODE QUALITY METRICS OF GENERATED CODE ACROSS DIFFERENT MODEL SIZES OF QWEN2.5-CODER-INSTRUCT, AT K = 1, 3, AND 5. **VPB** STANDS FOR VIOLATIONS PER BUILD.

| | | ***Qwen2.5-Coder-Instruct*** | | | | | |
|---|---|---|---|---|---|---|---|
| | | **0.5B** | **1.5B** | **3B** | **7B** | **14B** | **32B** |
| *k=1* | **VPB** | N/A | 3.06 | 2.31 | **1.00** | 1.05 | **1.00** |
| | **Critical VPB** | N/A | 0.47 | 0.15 | 0.00 | 0.00 | 0.00 |
| *k=3* | **VPB** | 2.00 | 2.81 | 3.68 | **1.00** | 1.09 | 1.06 |
| | **Critical VPB** | 0.00 | 0.42 | 0.91 | 0.00 | 0.00 | 0.03 |
| *k=5* | **VPB** | 4.00 | 3.37 | 2.62 | **1.09** | 1.12 | 1.11 |
| | **Critical VPB** | 1.00 | 0.61 | 0.31 | 0.00 | 0.02 | 0.05 |

parative performance of generic versus code-specific LLMs, and the influence of model size on code generation quality.

## A. Prompting Strategies

The results from Experiment 1 demonstrate that prompting strategies significantly influence the quality and buildability of generated code. Overall, it was observed that prompts with examples perform substantially better than prompts without examples. Zero-shot prompting consistently underperformed across all metrics, highlighting the limitations of relying solely on task descriptions without contextual examples. In contrast, few-shot and one-shot chain-of-thought (CoT) prompting techniques yielded superior results, particularly in match-based metrics such as CodeBLEU. These techniques leverage the in-context learning ability of the model, enabling it to better infer task structure and domain-specific patterns.

Interestingly, while CoT prompting was expected to enhance reasoning and structural coherence, its performance gains were marginal compared to standard few-shot prompting. In some cases, excessive reasoning chains may have introduced noise or led to overfitting on the examples, reducing generalization. This suggests that the benefits of CoT prompting may be task-dependent and that a single, well-crafted example may be more effective than multiple reasoning steps.

Upon manual inspection of the generated code, we observed that when prompts lack concrete examples of the desired coding style, the models tend to fall back on common patterns observed in their training data. For instance, generated methods often returned `nullptr` instead of raising exceptions or using optional objects, which contradicts the expected practice at ASML. Similarly, vectors were frequently constructed without pre-allocating memory, potentially impacting performance. These behaviors were especially prominent with zero-shot prompting, where the absence of guiding examples led to less aligned implementations. While the test coverage of the generated garages was low, manual review indicated that many of the garages without unit tests were functionally correct. This suggests that the reported metrics may underestimate the models' effectiveness in generating valid code.

## B. Generic versus Code-Specific LLMs

Experiment 2 revealed that code-specific LLMs generally outperform their generic counterparts, though the extent of this advantage varies across model families. For the Qwen2.5 models, the performance gap was minimal, indicating that the generic model already possessed strong code generation capabilities. However, in the Gemma and DeepSeek families, the code-specific models demonstrated improvements in both similarity and buildability metrics.

Manual inspection of the generated code revealed several notable limitations across the models. The generic Gemma model frequently reproduced the example garage from the prompt verbatim, suggesting limited generative capability. Both the code-specific Gemma model and the generic DeepSeek-V2 model often produced only pure virtual functions without any functional implementation, reflecting a lack of generalization and a shallow understanding of the task. This behavior also contributed to inflated build@k scores, as the code was syntactically valid but lacked meaningful functionality. In some cases, the generic DeepSeek-V2 model declined to generate code altogether, citing the task's complexity. Overall, the code-specific models showed a better grasp of the prompt, but still struggled to produce usable, fully implemented code.

## C. Impact of Model Size

Our findings confirmed that model size plays a critical role in the performance of LLMs for domain-specific code generation. Larger models consistently achieved higher similarity scores and generated more buildable code. The 14B and 32B models performed best overall, though the marginal gains between them suggest a reduced rate of improvement beyond a certain parameter threshold.

Interestingly, the 1.5B model achieved a high build@k score but often generated non-functional code consisting of virtual methods. This highlights a key limitation of using buildability as a proxy for correctness. Larger models, particularly those above 7B parameters, were more capable of generating functional code. Many garages generated by the 0.5B model lacked actual functionality and instead consisted of long lists of imports, a phenomenon known as the “repeat curse” [36], which is a common limitation of smaller-sized LLMs.

### *D. Threats to Validity*

Several threats to validity were identified across all experiments and can be grouped into three main categories: internal, external, and construct validity threats.

*1) Internal Validity Threats:* Prompt design bias arises because the structure, clarity, and phrasing of prompts may independently influence model performance. Although consistent prompt templates were used and reviewed for clarity, subtle differences in prompt formulation could still affect outcomes. Future work should consider systematically varying prompts to better understand and mitigate this bias. Furthermore, example selection bias affects prompting techniques that rely on examples to illustrate tasks, as the chosen examples may favor certain task types, coding styles, or reasoning approaches, potentially skewing model generalization. While we aimed to select representative examples aligned with the domain, expanding examples’ diversity or employing randomized sampling could improve robustness in subsequent studies.

*2) External Validity Threats:* Various threats to external validity limit the generalizability of our findings. Primarily, model dependence is a concern, as most experiments were conducted using a single or closely related family of LLMs. Since models vary significantly, our results may not transfer to other LLMs like GPT-4 or CodeLLaMa. Similarly, task dependence is a threat, as all code generation tasks were specific to the ASML domain, reflecting its unique software patterns and constraints. This specialization may limit the applicability of our findings to other industries. Lastly, a constant prompting strategy was used across all models and experiments, but different models may perform better with distinct prompting approaches. Future research should explore a broader range of models, tasks, and adaptive prompting strategies to enhance the generalizability of the findings.

*3) Construct Validity Threats:* We identified several threats to construct validity across the experiments conducted in this study. A key limitation is sparse unit test coverage, as many generated code artifacts lacked sufficient tests, restricting comprehensive verification of correctness; although manual inspection partially mitigated this, more extensive automated test suites would strengthen confidence in the results. Additionally, many of our metrics are binary indicators (e.g., pass/fail), which fail to capture partial correctness or subtle errors. Lastly, metric interdependence is a concern because certain metrics (e.g., unit test results, code quality) are only assessed on code that first builds successfully, introducing a filtering effect that biases downstream evaluations; designing evaluation pipelines to decouple or better account for these dependencies would improve the reliability of findings

## VII. Future Work

Several promising directions arise naturally from this work. First, the evaluation of generated code could be deepened by introducing human-in-the-loop assessments, where domain experts judge qualitative attributes such as readability, maintainability, and security. Complementary developer user studies would provide practical insights into usability, including productivity gains and levels of trust in generated code during real-world tasks.

The evaluation framework itself could be extended along two dimensions: (1) broadening unit test coverage across all garages and (2) incorporating system-level and integration testing to capture inter-component behavior and holistic functional correctness. On the modeling side, parameter-efficient fine-tuning techniques (e.g., LoRA) present an intriguing avenue for leveraging domain-specific data to determine whether measurable improvements can be achieved without the cost of full model retraining.

Another direction is the integration of agentic pipelines [37], enabling iterative, self-correcting workflows that could enhance reliability in domain-specific code generation. Similarly, the context retrieval strategy could benefit from retrieval-augmented generation, in which contextually relevant files are surfaced based on semantic similarity, potentially improving both precision and coherence of generated solutions.

Collectively, these directions offer a clear path to bridge research prototypes and industrial deployment of LLM-based code generation. Pursuing them requires dedicated experiments, user studies, and infrastructure that extend beyond the scope and objectives of our present collaboration, and we therefore leave them to future work.

## VIII. Conclusion

We investigated the use of LLMs for domain-specific code generation in the ASML leveling domain, examining three key aspects: prompting strategies, model specialization, and model size. First, we demonstrated that providing examples in prompts substantially improves code quality. Few-shot and one-shot chain-of-thought prompting consistently outperformed zero-shot prompting, highlighting the importance of guiding models with illustrative examples. Second, code-specific models generally surpassed their generic counterparts, with the most pronounced gains observed in the Gemma and DeepSeek-V2 families. Third, while increasing model size improved performance, the benefits tapered off at the higher end, raising important considerations about the cost–performance trade-off.

In all, despite the progress enabled by LLMs, ensuring functional correctness remains a key challenge in part due to limited unit test coverage and the tendency of smaller models to generate non-functional boilerplate. While we introduced build@k as a step toward addressing this challenge, tackling these limitations is essential for realizing reliable, domain-specific code generation in industrial settings.

## References

[1] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li *et al.*, "Large Language Models for Software Engineering: A Systematic Literature Review," *ACM Trans. Softw. Eng. Methodol.*, Sep. 2024, just Accepted. [Online]. Available: https://dl.acm.org/doi/10.1145/3695988

[2] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, "A Survey on Large Language Models for Code Generation," Jun. 2024, arXiv:2406.00515. [Online]. Available: http://arxiv.org/abs/2406.00515

[3] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo *et al.*, "Large Language Models for Software Engineering: Survey and Open Problems," in *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, May 2023, pp. 31–53. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10449667

[4] S. Joel, J. J. Wu, and F. H. Fard, "A Survey on LLM-based Code Generation for Low-Resource and Domain-Specific Programming Languages," Nov. 2024, arXiv:2410.03981 [cs]. [Online]. Available: http://arxiv.org/abs/2410.03981

[5] F. Zhang, B. Chen, Y. Zhang, J. Keung, J. Liu, D. Zan *et al.*, "RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation," Oct. 2023, arXiv:2303.12570 [cs]. [Online]. Available: http://arxiv.org/abs/2303.12570

[6] R. Koohestani, P. de Bekker, and M. Izadi, "Benchmarking ai models in software engineering: A review, search tool, and enhancement protocol," *arXiv preprint arXiv:2503.05860*, 2025.

[7] M. Izadi, J. Katzy, T. Van Dam, M. Otten, R. M. Popescu, and A. Van Deursen, "Language models for code completion: A practical evaluation," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[8] A. de Moor, A. van Deursen, and M. Izadi, "A transformer-based approach for smart invocation of automatic code completion," in *Proceedings of the 1st ACM International Conference on AI-Powered Software*, 2024, pp. 28–37.

[9] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Zurich, Switzerland: IEEE Press, Jun. 2012, pp. 837–847.

[10] M. Izadi, R. Gismondi, and G. Gousios, "Codefill: Multi-token code completion by jointly learning from structure and naming sequences," in *Proceedings of the 44th international conference on software engineering*, 2022, pp. 401–412.

[11] T. Van Dam, F. Van der Heijden, P. De Bekker, B. Nieuwschepen, M. Otten, and M. Izadi, "Investigating the performance of language models for completing code in functional programming languages: a haskell case study," in *Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering*, 2024, pp. 91–102.

[12] OpenAI, J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya *et al.*, "GPT-4 Technical Report," Mar. 2024, arXiv:2303.08774 [cs]. [Online]. Available: http://arxiv.org/abs/2303.08774

[13] A. Grattafiori, A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle *et al.*, "The Llama 3 Herd of Models," Nov. 2024, arXiv:2407.21783 [cs]. [Online]. Available: http://arxiv.org/abs/2407.21783

[14] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan *et al.*, "Evaluating Large Language Models Trained on Code," Jul. 2021, arXiv:2107.03374. [Online]. Available: http://arxiv.org/abs/2107.03374

[15] C. Team, H. Zhao, J. Hui, J. Howland, N. Nguyen, S. Zuo *et al.*, "CodeGemma: Open Code Models Based on Gemma," Jun. 2024, arXiv:2406.11409. [Online]. Available: http://arxiv.org/abs/2406.11409

[16] DeepSeek-AI, Q. Zhu, D. Guo, Z. Shao, D. Yang, P. Wang *et al.*, "DeepSeek-Coder-V2: Breaking the Barrier of Closed-Source Models in Code Intelligence," Jun. 2024, arXiv:2406.11931 [cs]. [Online]. Available: http://arxiv.org/abs/2406.11931

[17] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang *et al.*, "Qwen2. 5-coder technical report," *arXiv preprint arXiv:2409.12186*, 2024.

[18] X. Du, M. Liu, K. Wang, H. Wang, J. Liu, Y. Chen *et al.*, "Evaluating Large Language Models in Class-Level Code Generation," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, Apr. 2024, pp. 1–13. [Online]. Available: https://doi.org/10.1145/3597503.3639219

[19] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press *et al.*, "SWE-bench: Can Language Models Resolve Real-World GitHub Issues?" Nov. 2024, arXiv:2310.06770 [cs]. [Online]. Available: http://arxiv.org/abs/2310.06770

[20] H. Yu, B. Shen, D. Ran, J. Zhang, Q. Zhang, Y. Ma *et al.*, "CoderEval: A Benchmark of Pragmatic Code Generation with Generative Pre-trained Models," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, Feb. 2024, pp. 1–12. [Online]. Available: https://doi.org/10.1145/3597503.3623316

[21] A. Deshpande, A. Agarwal, S. Shet, A. Iyer, A. Kanade, R. Bairi *et al.*, "Class-Level Code Generation from Natural Language Using Iterative, Tool-Enhanced Reasoning over Repository," Jun. 2024, arXiv:2405.01573 [cs]. [Online]. Available: http://arxiv.org/abs/2405.01573

[22] G. Marvin, N. Hellen, D. Jjingo, and J. Nakatumba-Nabende, "Prompt Engineering in Large Language Models," in *Data Intelligence and Cognitive Informatics*, I. J. Jacob, S. Piramuthu, and P. Falkowski-Gilski, Eds. Singapore: Springer Nature, 2024, pp. 387–402.

[23] L. Binns, "How ASML is using software commonality to tackle complexity," Jan. 2024. [Online]. Available: https://www.linkedin.com/pulse/how-asml-using-software-commonality-tackle-complexity-lewis-dnthc

[24] C. Lambrechts, "Metrics for control models in a model-driven engineering environment," EngD Thesis, Sep. 2017, pDEng thesis.

[25] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large Language Models are Zero-Shot Reasoners," Jan. 2023, arXiv:2205.11916 [cs]. [Online]. Available: http://arxiv.org/abs/2205.11916

[26] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal *et al.*, "Language Models are Few-Shot Learners," Jul. 2020, arXiv:2005.14165 [cs]. [Online]. Available: http://arxiv.org/abs/2005.14165

[27] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia *et al.*, "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models," Jan. 2023, arXiv:2201.11903 [cs]. [Online]. Available: http://arxiv.org/abs/2201.11903

[28] Z. Yu, L. He, Z. Wu, X. Dai, and J. Chen, "Towards Better Chain-of-Thought Prompting Strategies: A Survey," Oct. 2023, arXiv:2310.04959 [cs]. [Online]. Available: http://arxiv.org/abs/2310.04959

[29] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a Method for Automatic Evaluation of Machine Translation," in *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, P. Isabelle, E. Charniak, and D. Lin, Eds. Philadelphia, Pennsylvania, USA: Association for Computational Linguistics, Jul. 2002, pp. 311–318. [Online]. Available: https://aclanthology.org/P02-1040

[30] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang *et al.*, "CodeBLEU: a Method for Automatic Evaluation of Code Synthesis," Sep. 2020, arXiv:2009.10297 [cs].

[31] C.-Y. Lin, "ROUGE: A Package for Automatic Evaluation of Summaries," in *Text Summarization Branches Out*. Barcelona, Spain: Association for Computational Linguistics, Jul. 2004, pp. 74–81. [Online]. Available: https://aclanthology.org/W04-1013

[32] L. Chen, Q. Guo, H. Jia, Z. Zeng, X. Wang, Y. Xu *et al.*, "A Survey on Evaluating Large Language Models in Code Generation Tasks," Mar. 2025, arXiv:2408.16498 [cs]. [Online]. Available: http://arxiv.org/abs/2408.16498

[33] S. Kulal, P. Pasupat, K. Chandra, M. Lee, O. Padon, A. Aiken *et al.*, "SPoC: Search-based Pseudocode to Code," Jun. 2019, arXiv:1906.04908 [cs]. [Online]. Available: http://arxiv.org/abs/1906.04908

[34] L. Jansen, "TomTom - Increasing Developer Productivity using the TiCS Framework." [Online]. Available: https://www.tiobe.com/knowledge/article/increasing-developer-productivity-using-the-tics-framework/

[35] P. Jansen, "Coding Standard Viewer." [Online]. Available: https://csviewer.tiobe.com/#/ruleset/intro?tagid=DIa5r7z2QqCEKHXmHLHfFw&setid=d4441hsNSnyvBQLpRWdAow

[36] J. Yao, S. Yang, J. Xu, L. Hu, M. Li, and D. Wang, "Understanding the Repeat Curse in Large Language Models from a Feature Perspective," May 2025, arXiv:2504.14218 [cs]. [Online]. Available: http://arxiv.org/abs/2504.14218

[37] A. C. Ionescu, S. Titov, and M. Izadi, "A multi-agent onboarding assistant based on large language models, retrieval augmented generation, and chain-of-thought," in *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, 2025, pp. 1208–1212.