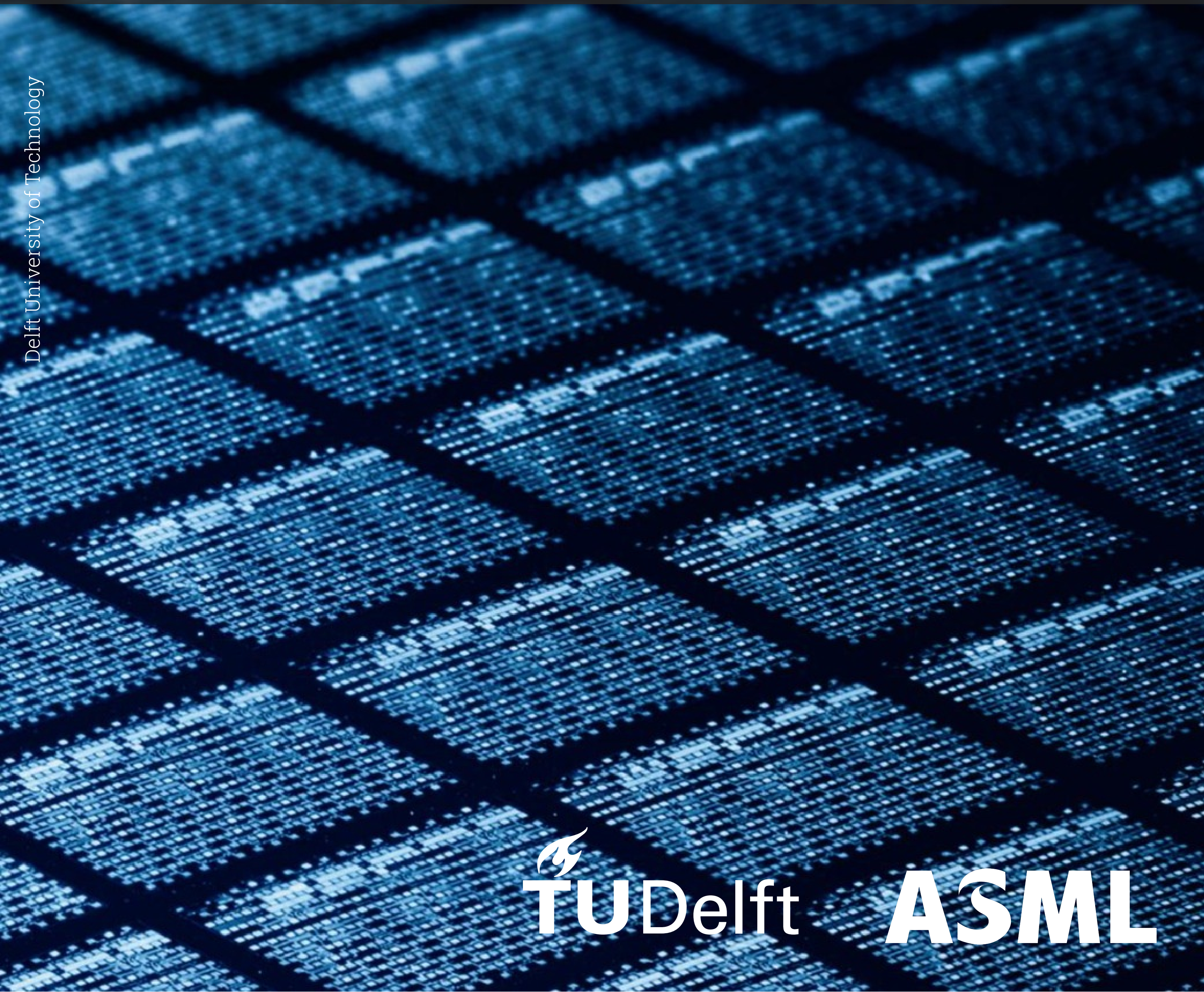
Gen-AI Meets Domain Expertise: LLMs for Domain Specific Code Generation

A study conducted at the ASML leveling department

Yash Mundhra

Delft University of Technology



Gen-AI Meets Domain Expertise: LLMs for Domain Specific Code Generation

A study conducted at the ASML leveling department

by

Yash Mundhra

to obtain the degree of Master of Science in Computer Science
at the Delft University of Technology,
to be defended on Thursday July 17, 2025 at 13:00.

Student number:	5017874	
Project duration:	September 15, 2024 – July 17, 2025	
Exam committee:	Asst. Prof. Dr. M. Izadi,	TU Delft, Daily Supervisor
	Prof. Dr. ir. F.A. Kuipers,	TU Delft, Thesis Advisor
	Asst. Prof. Dr. ir. U. Gadiraju,	TU Delft, Examiner
	MSc. CEng MIET Lewis Binns	ASML, Company Supervisor
	MSc. Max Valk	ASML, Company Supervisor

Preface

The past ten months have been a transformative and enriching experience. While this has been a very educational and rewarding journey, it would not have been possible without the help of many amazing people. I would like to express my deepest gratitude to everyone who has supported me throughout the journey of completing this thesis.

First and foremost, I would like to express my sincere appreciation to Maliheh Izadi at TU Delft for her unwavering guidance and expertise throughout my research. Her knowledge and insights have been instrumental in my understanding of AI for software engineering, and I've gained a wealth of knowledge in this area.

I am also grateful to Lewis Binns, Max Valk, Goran Brkic and the rest of the leveling department at ASML for providing me with the opportunity to conduct research with a practical industry aspect. Their guidance and support in navigating the ASML ecosystem and software stack have been a key component in my success. I appreciate the regular feedback sessions with my supervisors, which not only ensured the quality of my work but also fostered an enjoyable learning environment.

I would also like to extend my gratitude to Fernando A. Kuipers and Ujwal Gadiraju for serving on my thesis committee. Additionally, I appreciate the contributions of my fellow students and members of the AISE research group, who shared their insights and knowledge during the weekly reading clubs.

I would also like to express my gratitude to my friends for their support and friendship, which has been a constant source of comfort and motivation throughout my academic journey. From the long hours spent studying for exams to the celebrations after exam periods. Your friendship has been a treasure, and I'm so grateful to have you all in my life. Last but not least, I'd like to express my heartfelt appreciation to my family for their unconditional love and encouragement throughout this journey. Your presence has been a constant source of inspiration and motivation for me.

Yash Mundhra
Delft, July 2025

Abstract

Large Language Models (LLMs) have shown impressive performance in various domains, including software engineering. Code generation, a crucial aspect of software development, has seen significant improvements with the integration of AI tools. While existing LLMs have show very good performance in generating code for everyday tasks, their application in industrial settings and domain-specific contexts remains largely unexplored. This thesis investigates the potential of LLMs to generate code in proprietary, domain-specific environments, with a specific focus on the leveling department at ASML. The primary goal of this research is to assess the ability of LLMs to adapt to a domain they have not encountered before and to generate complex, interdependent code in a domain-specific repository. This involves evaluating the performance of LLMs in generating code that meets the specific requirements of ASML. To achieve this, the thesis investigates various prompting techniques, compares the performance of generic and code-specific LLMs, and examines the impact of model size on code generation capabilities. To evaluate the code generation capabilities of LLMs in repository-level scenarios, we introduce a new performance metric, build@k, designed to measure the effectiveness of generated code in compiling and building projects. The results showed that both prompting techniques and model size have a substantial influence on the code generation capabilities of LLMs. However, the performance difference between code-specific and generic LLMs was less pronounced and varied substantially across different model families.

Contents

List of Figures

List of Tables

List of Abbreviations

ASOME	ASML Software Modeling Environment
AST	Abstract Syntax Tree
BLEU	Bilingual Evaluation Understudy
CEM	Code Evaluation Metric
CNN	Convolutional Neural Network
CoT	Chain-of-Thought
DCA	Data Control and Algorithms
DTO	Data Transfer Object
EM	Exact Match
FFN	Feed Forward Network
FFT	Full Parameter Fine-Tuning
ICL	In-Context Learning
LLM	Large Language Model
LM	Language Modeling
MoE	Mixture-of-Experts
NLM	Neural Language Model
NLP	Natural Language Processing
NLU	Natural Language Understanding
PEFT	Parameter-Efficient Fine-Tuning
PLM	Pre-Trained Language Model
RAG	Retrieval-Augmented Generation
RNN	Recurrent Neural Network
ROUGE	Recall-Oriented Understudy for Gisting Evaluation
SCoT	Structured Chain-of-Thought
SE	Software Engineering
SLM	Statistical Language Model
SOTA	State of the Art

1 Introduction

LLMs have demonstrated remarkable capabilities in generating code across various programming languages and contexts, typically trained on extensive, publicly available datasets [36, 28, 22]. However, the challenge of generating code in domain-specific and repository-level environments remains largely unexplored [38, 71]. This research investigates the potential of LLMs to generate functional and maintainable code within the proprietary context of ASML's leveling department. The objective of this thesis is to assess how these models can adapt to a domain they have not explicitly encountered before, addressing the question: can LLMs effectively support developers by generating interdependent code in a closed, domain-specific repository?

Current research on LLM-based code generation predominantly focuses on straightforward, standalone code snippets or benchmarks using publicly available datasets [20, 57]. These scenarios are far removed from the intricate nature of industrial applications, where codebases are extensive, dependencies are complex, and domain-specific knowledge is crucial. Existing models excel at generating simple, modular units of code but fail when confronted with tasks requiring an understanding of domain-specific constraints and complex system-level interactions [38, 94]. In proprietary domains such as ASML, this challenge is further compounded as the models lack prior exposure to the company's unique coding patterns, frameworks, and conventions.

This research addresses this gap by focusing on the generation of code within the ASML leveling code base, a proprietary environment that involves interdependencies across various modules. The interdependent nature and specialized context of the ASML domain provides an ideal setting to understand how LLMs perform in such industrial situations. Unlike most studies that evaluate LLMs in contexts where training data aligns with testing scenarios, this work explores the application of LLMs in an environment where prior knowledge about the task is limited. By doing so, this study contributes to understanding how LLMs perform in specialized industrial contexts and their potential to generate coherent, functional code in situations they have not encountered before. This research aims to evaluate the feasibility of using LLMs in such environments, highlighting their strengths and limitations.

The main contributions made in this study are as follows:

- We study the application of LLMs for generating code in domain-specific settings, such as that of ASML, to investigate their potential in real-world scenarios.
- We introduce a new performance metric, termed *build@k*, which quantifies the success of generated code in compiling and building projects, providing a more comprehensive evaluation of project specific code generation capabilities.
- We construct a new code generation benchmark with ASML specific code, along with tests cases and context files. This benchmark enables the evaluation of LLMs for domain-specific code generation.
- We investigate various prompting techniques and their impact on performance when generating code for domain-specific environments, shedding light on the importance of tailored prompting

strategies.

- We compare the performance of generic LLMs and code-specific LLMs that have been fine-tuned on code datasets, aiming to understand the benefits of specialized models in code generation tasks.
- We examine the effect of model size on the performance of generated code, providing insights into the trade-offs between model complexity and code generation capabilities.

1.1. Research Questions

The application of LLMs for code generation has shown promise, but their effectiveness in proprietary, domain-specific environments, like the ASML leveling department, remains largely unexplored. To better understand the applicability of LLMs in such settings we consider various different aspects of LLM usage. We consider three key areas: the prompting strategy, the difference between generic and code-specific models and the impact of model size. These aspects are formally defined by the research questions outlined below, which together forms a structured study on the potential of LLMs to generate complex, interdependent code for ASML's leveling department.

- ***RQ1:* To what extent do different prompting techniques affect the performance of generated domain-specific code?**
  Prompting strategies, such as zero-shot, few-shot, and chain-of-thought prompting, play a significant role in shaping the outputs of LLMs. This question investigates the extent to which these 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 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 domain-specific code. Performance will be assessed using various metrics for functional correctness and code quality.
- ***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, but its impact on domain-specific code generation is not well understood. This question examines whether larger models, with their increased capacity for learning complex patterns, offer tangible benefits for generating domain specific code. By comparing models of varying sizes, this research question aims to determine the trade-offs between computational cost and performance in this specific context. Understanding this relationship is important for selecting the most efficient and effective model for industrial applications.

The experimental results revealed that the prompting techniques and the size of the model have quite a large impact on the quality of the generated domain-specific code. The performance difference between generic and code-specific models varied across different model families, with some demonstrating a more substantial difference in performance than others.

Specifically, our findings suggest that few-shot and one-shot chain-of-thought prompting generated the best performing code, while zero-shot prompting is hindered by the absence of prior examples to inform in the generation process. Regarding model size, our results reveal a general trend where larger models outperform smaller ones, although the rate of improvement is not consistent. Initially, the similarity of the generated code increases with model size, but this improvement slowed down after reaching 14B parameters. Finally, it was found that the code-specific models perform better than the generic models however, the difference in performance between the two models is not always equally large.

1.2. Thesis Outline

The subsequent sections in the report have been structured as follows. Chapter 2 provides the background knowledge relevant to the concepts employed in this thesis and discusses the related work that has been done previously on this topic. Chapter 3 details the design and methodology of the experiments we will be conducting to assess the performance of the LLMs in the domain-specific setting. Following this, Chapter 4 presents the results of the conducted experiments on the ASML leveling codebase. Chapter 5 analyzes and interprets these results, highlighting key insights and limitations of the conducted experiments. Finally, chapter 6 summarizes the key contributions of this thesis, explores directions for future research, and concludes the thesis.

2
Background & Related Work

2.1. Language Models

Language Modeling (LM) represents a foundational approach to enhancing machine language intelligence, with models structured to comprehend, generate, and predict human language patterns [98]. These models are integral to a broad spectrum of applications in Natural Language Processing (NLP) and Natural Language Understanding (NLU), supporting diverse tasks such as sentiment analysis [43], text summarization [95], and machine translation [24]. Although language modeling is a topic dating back to the 1950s [72] it has seen rapid development in the last few years.

Early language models were grounded in statistical approaches also known as *Statistical Language Models (SLMs)* [98]. The core idea behind SLMs is to predict the next word based on the most recent words using the Markov assumption. The Markov assumption states that the future state or event in a sequence depends only on the current state or event, and not on the past states or events [62]. Models with a fixed context length of size *n* are also called n-gram language models. While effective for small contexts, n-grams faced challenges with larger datasets and longer sequences due to the curse of dimensionality, the exponential growth in parameters needed to model more extended contexts. This limitation led to the development of *Neural Language Models (NLMs)* [98], which use neural networks to predict the probability of word sequences. NLMs achieve this by mapping words to low-dimensional embedding vectors and predicting the next word based on the combined context of the preceding words. Typically, NLMs are designed for specific tasks and trained on corresponding task-specific data, resulting in learned embedding spaces that are tailored to those tasks.

The next major advancement came with *Pre-Trained Language Models (PLMs)* [98], which are task-agnostic. PLMs generally use the "pre-training and fine-tuning" paradigm, where models with architectures such as Recurrent Neural Networks (RNNs) or transformers are trained on vast amounts of data and then fine-tuned to specific tasks using small amounts of task-specific data. This pre-training and fine-tuning approach revolutionized NLP by providing models that could be adapted to a wide range of tasks with minimal data. Building on this, *LLMs* [98], such as GPT-4 and LLaMa, represent scaled versions of PLMs, often comprising billions of parameters. These models leverage vast computational resources and datasets, enabling them to achieve state-of-the-art performance in various applications.

2.2. LLMs in Software Engineering

LLMs have shown the ability to replicate human linguistic skills, bringing about numerous transformations across various professional fields. One area that has particularly benefited from the advancements of LLMs is *Software Engineering (SE)* [28]. The application of LLMs spans the entire software engineering life cycle, from initial requirement engineering to final deployment. In this section, we dive deeper into the application of LLMs in the "code implementation" phase of the SE life cycle.

The pivotal research by Hindle et al. [26] proved that software, although theoretically complex, is actually quite predictable through elementary 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.

In the context of Software Engineering, a distinction can be made between general-purpose LLMs and code-specific LLMs. General-purpose LLMs, such as GPT-4 [63] and LLaMa 3 [25], 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 [36]. By focusing on programming-related tasks and challenges, these code-specific models have achieved even better performance than generic LLMs, in terms of functional correctness.

2.2.1. Code-Specific LLMs

In 2021, Chen et al. [13] presented CodeX, a code-specific language model fine-tuned from the GPT-3 model family [9]. The fine-tuning dataset consisted of 159GB publicly available code scraped from GitHub repositories. The GPT family of models follow the decoder-only architecture (see 2.3.3) with transformers forming the backbone of the model. Each layer has multi-head self-attention mechanisms and feed-forward neural networks, enabling the model to predict the next word in a sequence based on context. It is trained auto regressively, predicting each token in a sequence based on the previous tokens, allowing it to generate coherent and contextually relevant text.

In order to evaluate the CodeX model, the study also introduced HumanEval, a novel evaluation dataset designed to assess functional correctness in code generation from docstrings. Following the publication of this work, research on LLM-based code generation surged, with many subsequent studies utilizing HumanEval for benchmarking.

In 2024, Zhao et al. [80] introduced CodeGemma, a decoder-only, code-based LLM that builds upon Google's Gemma models, adhering to the same architectural design. The CodeGemma models underwent further training on a vast dataset of 500 billion tokens, comprising primarily of code with smaller proportions of natural text. This fine-tuning of the core Gemma model enabled CodeGemma to achieve State of the Art (SOTA) performance in both code generation and completion tasks. CodeGemma had a better performance than its generic counterpart on the HumanEval problems, demonstrating its enhanced capabilities.

In total the Google team released three models for CodeGemma:

- **2B base model:** specialized in infilling and open-ended generation.
- **7B base model:** trained with both code infilling and natural language.
- **7B instruct model:** allows users to engage in conversational interactions about code.

In the same year that CodeGemma was introduced, the DeepSeek-AI team released DeepSeek-Coder-V2 [15] an open-source Mixture-of-Experts (MoE) code language model. The DeepSeek-Coder-V2 model was further pre-trained from a checkpoint of DeepSeek-V2 [16] with 6 trillion additional tokens. As a result of the continued pre-training, the coder model significantly enhanced the coding and mathematical reasoning capabilities of the generic model, while still maintaining similar performance in general language tasks. The authors stated that the coder model demonstrated superiority over all open source models while performing on par with the leading closed-source models such as GPT-4 Turbo and Gemini 1.5 Pro.

The authors released a total of four models for DeepSeek-Coder-V2:

- **DeepSeek-Coder-V2-Base:** A 236B parameter model for general purpose code understanding and generation.
- **DeepSeek-Coder-V2-Instruct:** An instruction tuned version of the base model for interactive coding tasks.
- **DeepSeek-Coder-V2-Lite-Base:** A 16B lightweight version of the base model optimized for efficiency

- **DeepSeek-Coder-V2-Lite-Instruct:** A tuned version of the lite model for code-related dialogue.

Slightly after the release of DeepSeek-Coder-V2, Hui et al. [31] released Qwen2.5-Coder, a decoder-only LLM developed by Alibaba. The Qwen2.5-Coder series represents a substantial upgrade over its predecessor, CodeQwen1.5. Built upon the architecture of Qwen2.5, the coder model was further pre-trained on an extensive dataset of over 5.5 trillion tokens of code-centric data, including repositories and documentation. As a result, the coder model achieved significant performance gains compared to the generic Qwen2.5 model on code-related tasks.

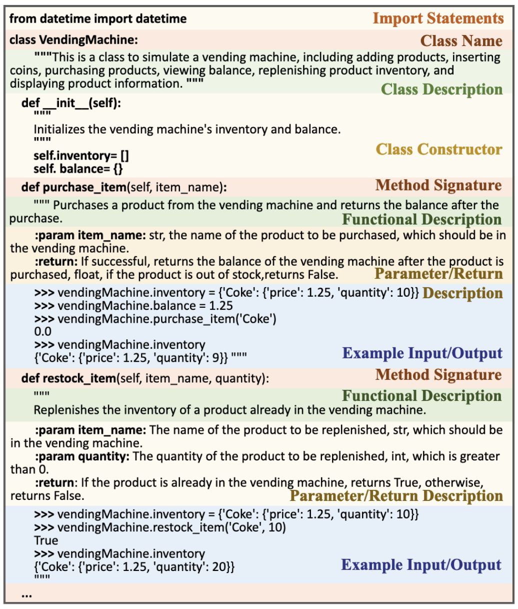
The Qwen2.5-Coder family consists of two main variants that are released in six sizes (0.5B, 1.5B, 3B, 7B, 14B, and 32B parameters):

- **Base Models:** a foundational model for developers to fine-tune their own models.
- **Instruct Models:** official aligned model that can be used for chat directly.

2.2.2. Domain Specific Code Generation

While LLMs have demonstrated strong capabilities in code generation, current benchmarks, such as HumanEval, focus on relatively simple scenarios. These benchmarks primarily evaluate the ability of LLMs to generate individual units of code, such as standalone functions or statements. In essence, software development comprises of broader contextual dependencies and multiple interdependent units of code [20]. Jimenez et. al. [37] evaluated the performance of LLMs in a "realistic software engineering setting", where the models are tasked to resolve issues submitted to popular GitHub repositories. The best performing model, Claude 2, was only able to resolve 1.96% of the issues. Due to the poor performance of these models in domain-specific code generation settings, some researchers have begun exploring this problem, but the field remains largely underexplored.

To evaluate the performance of LLMs in more realistic and challenging scenarios, Du et al. [20] introduced a novel code generation benchmark called ClassEval. ClassEval tasks involve an input description detailing the class to be generated, a corresponding test suite, and a benchmark solution. To ensure the generated code aligns with a consistent interface specified by the test suite, the benchmark also includes a *class skeleton*, which serves as a blueprint for the target class. This skeleton provides both class-level and method-level information, guiding the structure and functionality of the generated code. Figure 2.1 shows an example of the skeleton in the ClassEval benchmark. Overall, it was observed that the performance of existing SOTA LLMs declined significantly on the class-level benchmark, underscoring the need for further research and advancements in this area.


Import Statements
Class Name
Class Description
Class Constructor
Method Signature
Functional Description
Parameter/Return
Description
Example Input/Output
Method Signature
Functional Description
Parameter/Return Description
Example Input/Output

```
from datetime import datetime
class VendingMachine:
    """This is a class to simulate a vending machine, including adding products, inserting
coins, purchasing products, viewing balance, replenishing product inventory, and
displaying product information. """
  def __init__(self):
    """
    Initializes the vending machine's inventory and balance.
    """
    self.inventory= []
    self. balance= {}
  def purchase_item(self, item_name):
    """ Purchases a product from the vending machine and returns the balance after the
purchase.
    :param item_name: str, the name of the product to be purchased, which should be in
the vending machine.
    :return: If successful, returns the balance of the vending machine after the product is
purchased, float, if the product is out of stock,returns False.
    >>> vendingMachine.inventory = {'Coke': {'price': 1.25, 'quantity': 10}}
    >>> vendingMachine.balance = 1.25
    >>> vendingMachine.purchase_item('Coke')
    0.0
    >>> vendingMachine.inventory
    {'Coke': {'price': 1.25, 'quantity': 9}} """
  def restock_item(self, item_name, quantity):
    """
    Replenishes the inventory of a product already in the vending machine.
    :param item_name: The name of the product to be replenished, str, which should be
in the vending machine.
    :param quantity: The quantity of the product to be replenished, int, which is greater
than 0.
    :return: If the product is already in the vending machine, returns True, otherwise,
returns False.
    >>> vendingMachine.inventory = {'Coke': {'price': 1.25, 'quantity': 10}}
    >>> vendingMachine.restock_item('Coke', 10)
    True
    >>> vendingMachine.inventory
    {'Coke': {'price': 1.25, 'quantity': 20}}
    """
  ...
```

Figure 2.1: Example of class skeleton in the ClassEval benchmark providing class-level and method-level information [20].

While the ClassEval benchmark represents an important contribution, the problems it includes can still be viewed as standalone, treating each class as an isolated unit. The only dependencies these classes rely on are common libraries, which are likely to be a part of the LLMs training data. In contrast, real-world classes often depend on other components within the same repository, a complexity not captured in this dataset. To address this limitation, Yu et. al [92] proposed the CoderEval benchmark. This dataset constructed from real world projects makes it possible to evaluate the LLMs on "pragmatic code generation", where the generative ability is evaluated using non-standalone functions.

However, CoderEval still faces limitations, as its scope remains confined to line- and function-level tasks within a repository's context. To address these limitations, Deshpande et al. [17] introduced the RepoClassBench benchmark. This dataset evaluates LLMs on the generation of non-standalone classes within the context of a repository, offering a more realistic reflection of real-world scenarios.

2.2.3. Context Retrieval Techniques

One effective way to enhance the performance of LLMs in repository-level code generation is to provide relevant context within the prompt, allowing the model to better infer the desired code output. Retrieval-Augmented Generation (RAG) [46] is one of the methods through which LLMs can incorporate contextual information. RAG-based models involve an initial step where an external data source, such as a code repository, is accessed to retrieve relevant information, which is then used to generate a more informed response. This approach makes LLMs more practical for real-world applications, because it ensures that the generated output is grounded in retrieved evidence, leading to more reliable and accurate results [5].

When a LLM is provided with examples and other relevant context in the prompt it shows the In-Context Learning (ICL) ability, where it is able to learn from a few examples in the context [18]. Formally defined "ICL is a paradigm that allows language models to learn tasks given only a few examples in the form of demonstration" [18]. Using RAG and ICL, models can identify and learn from patterns embedded in the context, making more accurate predictions and eliminating the need for traditional supervised learning, which requires updating model parameters.

Li et al. [47] proposed AceCoder, a RAG based LLM that retrieves code snippets with similar descriptions to the one it aims to generate. This approach is grounded in the hypothesis that, in real-world scenarios, developers typically search for similar programs when faced with a new requirement, and then either learn from them or directly reuse relevant content. The retrieval process utilizes the BM25 metric [68] to estimate the lexical similarity between two sentences, with higher BM25 scores indicating greater similarity. Ultimately, the most relevant "similar programs" are selected and provided, along with the prompt, as contextual information to inform the generation process.

RepoCoder [94] builds upon the concept of using repository-level RAG for code completion by introducing an iterative pipeline. This pipeline leverages the generated code to refine the retrieval process, enhancing the overall performance. The retrieval process begins by formulating a query based on the unfinished code, which is then used to retrieve the most similar code snippets. To further improve the retrieval context, the query is augmented with previously generated code in subsequent iterations, which provides valuable additional information that improves the retrieval process. The iterative RAG process employed by the RepoCoder framework is illustrated in Figure 2.2.

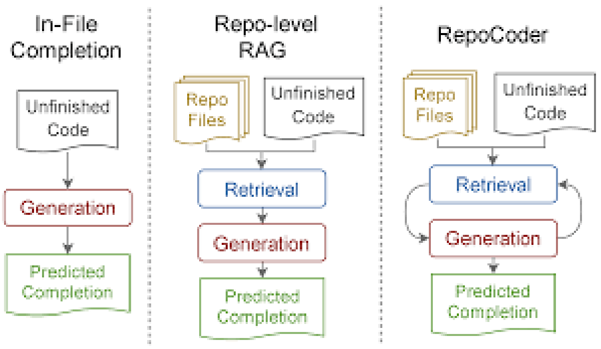


Figure 2.2: Iterative RAG process as proposed by the RepoCoder framework [94].

Zhang et al. proposed a LLM-based agent framework, *CodeAgent* [96], which leverages various external sources to enhance the problem-solving capabilities of LLMs in repository-level code generation tasks. The framework utilizes a range of external programming tools, including website search, documentation reading, code symbol navigation, format check, and code interpreter, to support the code generation process. By incorporating these tools, the *CodeAgent* framework aims to simulate the workflow of a human developer, who typically gathers relevant knowledge, adapts existing code to meet specific requirements, and verifies programs using various tools.

2.2.4. Hallucination

LLMs have demonstrated impressive capabilities in both natural language generation and code generation, but one of the significant challenges they still face is hallucination. Ji et al. [35] defined hallucination in the natural language generation context as "the generated content that is nonsensical or unfaithful to the provided source content". For LLMs focussed on generating code, hallucination can be defined as "the generated code that has one or more code defects such as dead or unreachable code, syntactic or logical errors, robustness issues such as the code fails on edge cases or raises an exception, or has security vulnerabilities or memory leaks" [2].

Liu et al. [54] provided an in-depth analysis of hallucinations that occur in LLM-powered code generation. The taxonomy conducted in the study consists of 5 primary categories and 19 types of hallucinations as shown in figure 2.3. The five main categories of hallucinations that were identified were:

- **Intent Conflicting:** Refers to a situation where the generated code deviates from the user's intent with a relatively small semantic correction.
- **Inconsistency:** When the generated code snippet is not consistent with the context leading to minor logical issues.
- **Repetition:** Excessive repetition of some code snippet in the generated content leading to bad code quality.
- **Dead Code:** Refers to snippets of code that are executed but its result is never utilized in any other computation.
- **Knowledge Conflicting:** Occurs when the generated code contains inconsistencies, such as using incorrect variables or invoking APIs in an incorrect manner.

The results showed that the most common types of code hallucinations is “intent conflicting” and “context inconsistency”, followed by “context repetition” and "knowledge conflicting”. The "dead code" category was observed with the lowest frequency.

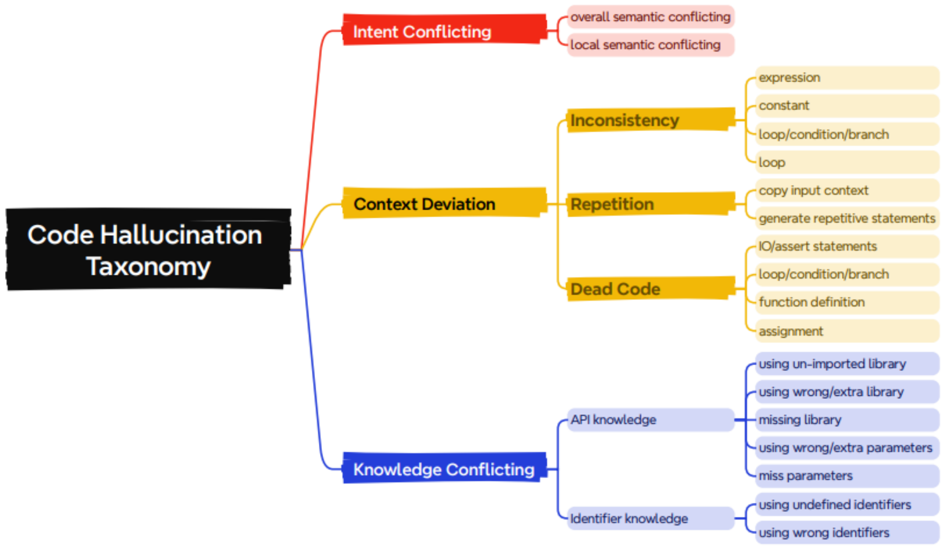


Figure 2.3: Hierarchical Taxonomy of hallucinations in LLM-generated code [54].

2.2.5. Sampling Parameters

The process by which a large language model selects the next token has a large impact on the quality of the generated output. Specifically, the model assigns a set of logits to each candidate word, representing its likelihood of being chosen as the next output. These logits are then transformed into a probability distribution through the application of a softmax function, which normalizes the values to ensure they sum to 1. This resulting probability distribution serves as the basis for selecting the next token, with each word's probability reflecting its relative likelihood of being chosen.

Always picking the highest-probability token from the next-word distribution computed by the model often produces dull and repetitive text. On the contrary, unrestricted random sampling from the full distribution can lead to incoherent gibberish [66]. To balance fluency and diversity in the output, most LLMs rely on decoding/sampling parameters that control the tradeoff between determinism and randomness. The four key parameters that are used by a majority of LLMs are *temperature (T)*, *top-k*, *top-p*, and *repetition penalty*. This section explains the purpose of each parameter and the influence it has on the generated text.

Temperature

Temperature T is a parameter that smoothens or sharpens the probability distribution over the next-token candidates [98]. A lower value for T makes the tokens with the highest-probability more likely to be selected. A high value for T flattens the distribution allowing for more randomness in the next token selection. Mathematically, this is achieved by rescaling the model's logits through division by T before applying the softmax function, which has the effect of altering the relative probabilities of the candidate tokens.

In practice, adjusting the value for T has a large impact on the output diversity and creativity versus stability. At $T \to 0$ the model is decoding in a greedy way by always choosing the most probable token and at a very high T the choice can become almost random, degrading the fluency and correctness of the model. A research by Chen et al. [13] discovered that the temperature coefficient T also has a significant influence on the code generation results of LLMs. Increasing the T value increases the chances of generating a correct solution (by exploring more approaches) but also introduces more mistakes in the output.

Liu et al. [55] investigated the effects of the T value on the code generation capabilities of the LLMs. It was found that a lower temperature value tends to perform better for smaller k, while a higher temperature works better for larger k, where k represents the amount of times code is being generated for the same task.

In response to the challenges of selecting an optimal temperature parameter, Zhu et al. [100] proposed a novel approach that adapts the temperature in real-time during code generation. They authors observed that code has parts that are *"challenging tokens"* (often the beginning where many choices exist) and later parts that are *"confident tokens"*. They suggest using a higher value for T for the initial uncertain segments and a lower T for predictable segments. This approach improved code generation accuracy proving that varying the T value in a single generation can have a valuable impact.

Top-p
Top-p sampling, also known as nucleus sampling [27], is a decoding strategy that aims to truncate the token-set using a probability driven approach. It was proposed to *"truncate the unreliable tail"* of the distribution filtering out the low probability candidates. During each generation step, the model ranks all possible tokens by their predicted probabilities and selects the smallest subset of tokens that cumulatively exceed a predetermined probability threshold p. The next token is sampled from this set of *high probability tokens*, with all other tokens excluded. Overall it was found that utilizing top-p for decoding yields text that is more natural and diverse without sacrificing fluency and coherence. In practice, many SOTA LLMs and also coding models use a combination of top-p sampling with a tuned temperature parameter. This combinations means that the model first samples from a softened distribution (using temperature) and then truncates to the nucleus. The net effect is that the model can generate many diverse yet plausible code snippets. Chen et al. [13] also highlighted that using this combination increases the chance of finding correct solutions in at least one of the outputs (improving the pass@k score).

Top-k
Top-k sampling, as introduced by Fan et al. [21], operates on a similar principle to top-p sampling, where the model's choices are confined to a fixed-size subset of the most probable tokens. In the top-k approach, at each decoding step, the model identifies the k tokens with the highest probabilities and sets the probabilities of all other tokens to zero, effectively eliminating them from consideration. The next token is then sampled exclusively from this top-k list. However, a key limitation of top-k sampling is its rigid cutoff, which can be insensitive to context. In scenarios where multiple tokens exhibit similar probabilities, resulting in a flat distribution, a small k value may unnecessarily exclude viable words. Top-p sampling has been found to perform better in practice, as it dynamically adjusts the number of considered options based on the probability distribution, allowing for more nuanced and contextually appropriate word selections.

Repetition Penalty
The repetition penalty [39] is a decoding-time heuristic used to prevent the model from generating the same token or sequence repeatedly. Formally this is achieved by dividing the logit of a token, that has appeared in the output, by a constant factor *alpha* (repetition penalty value) before the next token is sampled. As a result, the same token will have a lower probability of being chosen again, given that $alpha > 1$. Liu et al. [56] conducted an empirical study to investigate the nature of repetition across SOTA code LLMs and compared various repetition mitigation techniques. It was found that repetition penalty also has a significant impact on the functional correctness of the generated code. A penalty of 1.2 on the DeepSeekCoder-1.3b-I model reduces repetition from 29.1 to 0.5 but simultaneously also dropped the Pass@1 accuracy from 18.5 to 0.2. This is likely because the model tries to avoid reusing crucial tokens such as previously defined variables or a necessary function call. The authors instead proposed a technique known as DeRep that significantly outperformed standard repetition mitigation techniques. In practice the repetition penalty is often combined with other parameters such as the top-p.

2.3. LLM Architectures

2.3.1. Transformers

The transformer architecture as proposed by Vaswani et. al. [82] is the most popular LLM architecture and serves as the backbone of almost all current SOTA models and is also often the building block, of other LLM architectures such as encoder-only, decoder-only and encoder-decoder. At the heart of the transformer architecture lies the (self-)attention mechanism which is able to capture long-range dependencies much more efficiently as opposed to the recurrence and convolution mechanism in RNNs and CNNs respectively.

The original transformer architecture that was proposed in the paper consisted of both an encoder and decoder. The encoder consists of a stack of $N = 6$ identical layers consisting of two sublayers. The first sublayer implements a multi-head self-attention mechanism and the second is a fully connected *Feed Forward Network (FFN)*. The decoder also consists of a stack of $N = 6$ identical layers, but with three sublayers. The first sublayer receives the previous output and implements multi-head self attention over it. The second layer performs multi-head self attention over the output of the encoder stack. The third layer is a fully connected FNN. Figure 2.4 displays the original transformer architecture.

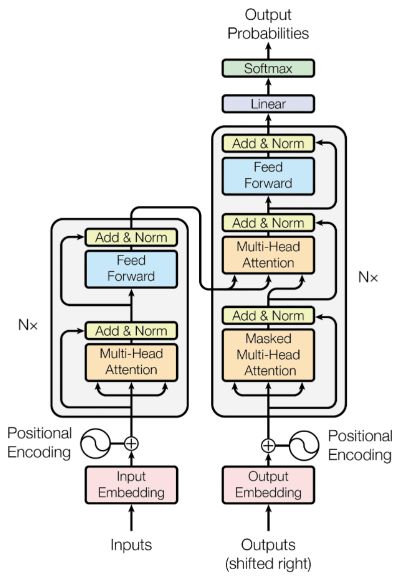


Figure 2.4: The encoder-decoder structure of the transformer architecture as proposed by Vaswani et. al. [82]

(Self-)Attention Mechanism

The core idea behind the transformer model is the attention mechanism that was first introduced in the paper by Bahdanau et. al. [6]. The attention mechanism allows a model to learn how to relate input words to other words through three weight matrices: query matrix W_q, key matrix W_k, value matrix W_v. Each input token x_i is converted into a query, a key, and a value vector, by multiplying the embedding vector with the weight matrices. The attention scores are computed by taking the scaled dot product of the query and key vectors followed by a normalization step using softmax. Finally, the attention output is computed as a weighted sum of the value vectors. The final output vector is a representation that captures the contextual relationships of tokens in the input, even across long sequences.

2.3.2. Encoder-Only

Encoder-only LLMs only use the encoder module to process and encode input into its hidden representation. The encoder module comprises multiple layers of multi-head attention and feed-forward networks [82], enabling it to capture dependencies and extract features across positions in an input sequence. This layered design effectively handles long-range dependencies. In the field of SE, models that follow the encoder-only architecture such as CodeBERT [23], CodeRetriever [51], and BERTOverflow [78] often excel in tasks that require enhanced understanding of the code snippet for tasks such as code reviews and bug reporting.

2.3.3. Decoder-Only

As the name suggests, decoder-only models only consist of a decoder module that directly generates an output sequence based on a given context or input. The output in decoder-only models is generated token-by-token where each token generated by the decoder is conditioned on the previous tokens generated and the context. These models excel in few-shot learning scenarios without adding prediction heads or fine-tuning but can face limitations in understanding context deeply. For software engineering tasks decoder models are mainly used for generative task such as code generation and code completion.

Some specialized models such as CodeGeeX [99], StarCoder [49], and Codex [13] are being used for SE applications.

2.3.4. Encoder-Decoder

Encoder-decoder LLMs consists of both an encoder and decoder module. The encoder encodes the input into a hidden space while capturing the underlying structure and semantics. The decoder utilizes the hidden mapping to generate the target output text, translating the abstract representation into concrete and contextually relevant expressions. SE specific models such as CodeT5+ [84], AlphaCode [52] and CodeFusion [75] demonstrate the architecture's adaptability to various SE tasks such as code summarization, code translation and program repair.

2.4. Evaluation Metrics

Software quality has long been a critical focus in software engineering, so it comes as no surprise that recent years have seen extensive research dedicated to evaluating code generated by LLMs. There are two main types of prevailing Code Evaluation Metrics (CEMs), that asses the performance of a generated piece of code: **match-based** and **execution-based** metrics [19]. Section 2.4.1 provides an overview of the most common match-based metrics, while section 2.4.2 presents an overview of the most common execution-based metrics.

2.4.1. Match-based metrics

Match-based metrics evaluate the similarity between the generated code and a reference implementation by comparing tokens, syntax structures, or exact outputs. Examples of commonly used match-based CEMs include Exact Match (EM), BLEU [64], ROUGE [53], and CodeBLEU [67] as described below.

Exact Matching Accuracy

The exact match accuracy measures the percentage of code snippets that exactly match the reference code [58]. The straightforward yet stringent nature of this method makes it a good tool for measuring code generation accuracy particularly for evaluating code completion models.

Bilingual Evaluation Understudy (BLEU)

The BLEU metric, first proposed by Papineni et. al. [64], is a metric originally used for machine translation evaluation. The BLEU score is calculated based on the number of matching n-grams between the translated text and the reference text. It then calculates the weighted mean of the of these matches to get the overall similarity score. For code generation BLEU is often used to evaluate the similarity between the reference and generated code. There, are also some disadvantages to using BLEU for code comparison. Firstly, the syntax and semantic nature of code is different to natural language due to which it is not a very accurate measure of code similarity. Additionally, code often contains a large number of unique identifiers such as variable and function names which may completely differ between two blocks of code even though the functionality is exactly the same.

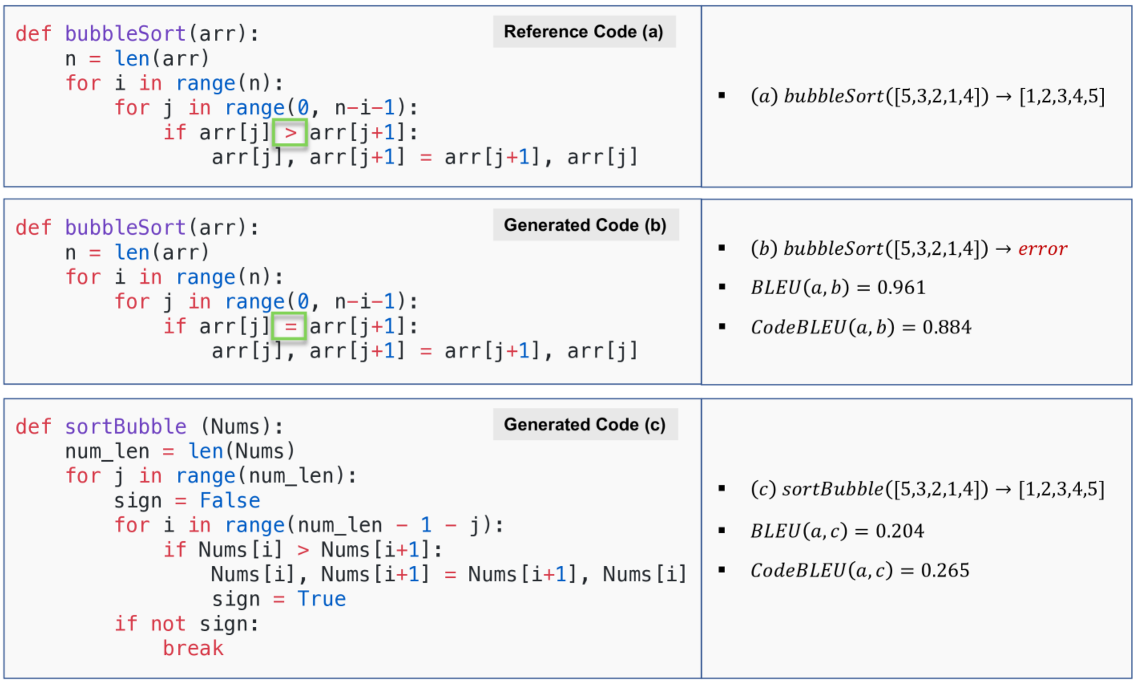
CodeBLEU

To address the limitations of BLEU, Ren et al. [67] introduced CodeBLEU, an extension of the BLEU metric. CodeBLEU enhances the traditional n-gram matching approach by incorporating syntactic and semantic information specific to code. This is achieved by leveraging Abstract Syntax Trees (ASTs) to represent code syntax and data-flow to capture code semantics. The final CodeBLEU score is calculated by combining the scores of four components; n-gram match, weighted n-gram match, syntax match, and data flow match, in a weighted manner. By considering all four aspects, CodeBLEU not only assesses the similarity between code blocks but also gains insight into their internal logic and functionality to a certain extent, thereby providing a more accurate comparison. The experimental results from the original paper demonstrate that CodeBLEU exhibits a stronger correlation with programmer-assigned scores compared to BLEU and accuracy, indicating its effectiveness in evaluating code similarity as opposed to BLEU.

Recall-Oriented Understudy for Gisting Evaluation (ROUGE)

ROUGE is another evaluation metric that was traditionally created for comparing computer-generated summaries to those created by humans [53], however it has since been adapted for code generation evaluation. The metric is computed by taking 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 but fails to evaluate functional correctness.

Despite their extensive usage, match-based metrics still have limitations in assessing functional correctness, since they cannot determine whether the generated code is functionally equivalent to the reference code. For example in Figure 2.5 we observe that code (b) has a much higher similarity to our reference code (a), than (c). However upon execution we notice that code (b) is functionally incorrect and does not even compile, while (c) is a different rendering of the same function. As a result we have observed that measuring similarity alone is not sufficient for code evaluation and we must use execution-based metrics.



Reference Code (a)

```
def bubbleSort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
```

- (a) *bubbleSort*([5,3,2,1,4]) → [1,2,3,4,5]

Generated Code (b)

```
def bubbleSort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] = arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
```

- (b) *bubbleSort*([5,3,2,1,4]) → *error*
- $BLEU(a, b) = 0.961$
- $CodeBLEU(a, b) = 0.884$

Generated Code (c)

```
def sortBubble (Nums):
    num_len = len(Nums)
    for j in range(num_len):
        sign = False
        for i in range(num_len - 1 - j):
            if Nums[i] > Nums[i+1]:
                Nums[i], Nums[i+1] = Nums[i+1], Nums[i]
                sign = True
        if not sign:
            break
```

- (c) *sortBubble*([5,3,2,1,4]) → [1,2,3,4,5]
- $BLEU(a, c) = 0.204$
- $CodeBLEU(a, c) = 0.265$

Figure 2.5: Evaluating generated code using BLUE and CodeBLEU. Similarity based metrics score the actual functional correct code (c) lower than the incorrect code (b) [19].

2.4.2. Execution-based metrics

To overcome the limitations of similarity-based metrics, particularly in assessing the functional correctness of code, execution-based metrics have been developed. These metrics involve compiling and executing the code block to evaluate its behavior and output. The remainder of this section provides an overview of the most prominent execution-based evaluation metrics, which offer a more comprehensive assessment of code quality and functionality.

Pass@k

Kulal et. al. [42] introduced an evaluation metric known as pass@k. The pass@k metric is calculated by generating k possible code solutions for each task, and then reporting the fraction of tasks that are successfully solved. A problem is deemed as solved if any of the sample passes all the associated tests. This way of measuring pass@k was found to exhibit high variance thus Chen et al. [13] proposed to use an unbiased estimator for computing pass@k.

It is defined as:

$$\text{pass@k} := \mathbb{E}_{\text{task}} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right]$$

Where: n is the total number of sampled candidate solutions, k is the number of randomly selected code solutions, with $n \geq k$, and c is the count of correct samples within the k selected.

Although execution-based CEMs that utilize unit tests can effectively assess the functional correctness of the generated code, they have a significant limitation: their reliability is heavily dependent on the quality and comprehensiveness of the unit tests. In practice, providing a sufficient and diverse set of test cases for each code snippet can be a substantial burden and is not always feasible, which can compromise the accuracy and robustness of the evaluation results.

Code Quality
Although LLM-generated code may produce functionally correct results, it can still contain undesirable characteristics such as *code smells* and *vulnerabilities*. A study by Siddiq et al. [74] revealed that the code snippets used to train LLMs often lack rigorous quality control, failing to adhere to "coding standards, implementing proper design decisions, and using secure (defensive) coding practices" As a consequence, these training sets can contain code smells, which in turn can compromise the generation capabilities of the models [73]. In [73], the authors propose *FRANC*, a lightweight framework for recommending more secure and high-quality source code from LLMs. The framework utilizes tools such as Bandit or SpotBugs to rank code snippets based on a configurable quality score and feeds the analysis of the code snippets back to an LLM for fixing. Figure 2.6 presents an overview of the proposed framework.

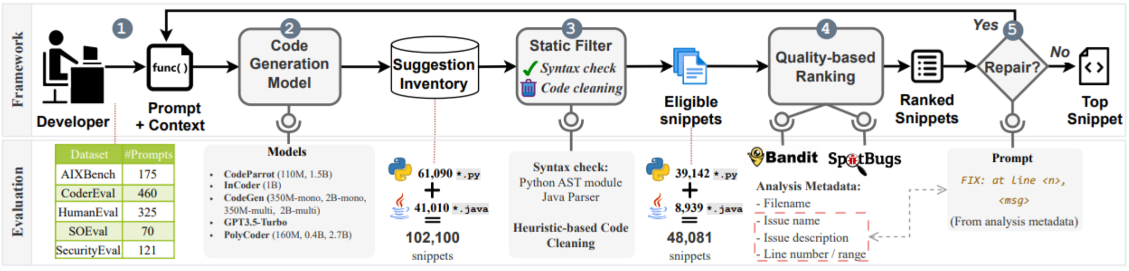


Figure 2.6: *FRANC* framework used for producing more secure and high quality code from LLMs [73].

2.5. Prompt Engineering

In LLMs, a prompt refers to the textual input supplied by users to steer the models' output. The quality of the prompt significantly impacts the performance of many LLMs [61]. The art and science of creating well-structured prompts to maximize the capabilities of LLMs is known as *prompt engineering*. A good prompt typically incorporates three main elements: task description, input data, and contextual information [98]. A *task description* is a specific instruction that LLMs are expected to follow, prompts should clearly describe the goal in natural language. In most cases, *input data* can be easily described in natural language. For special input data such as graphs or tables a transformation may be required to make it readable for LLMs. Finally, *contextual or background information* is often also important, as it provides the model with additional details or situational awareness necessary for generating accurate and relevant responses.

The following sections describe a few of the most notable prompt engineering approaches.

2.5.1. Zero- & Few-Shot Prompting

Zero-shot prompting [41] involves guiding a model to perform a task without any prior explicit training on that specific task. Instead, carefully designed prompts are used to direct the language model towards the new task. In this approach, the prompt contains only a task description, with no explicit input-output examples provided. The model relies on its pre-existing knowledge to generate predictions as accurately as possible. Few-shot prompting [9], in contrast, involves providing models with a small number of input-output examples to help them better understand a given task. Even a handful of well-crafted examples can significantly enhance the model's performance on complex tasks. This approach has some limitations. First, it increases the length of the input sequence, which can impact efficiency. Second, the selection of examples plays a crucial role, as it can heavily influence the model's behavior and output quality.

2.5.2. Chain-of-Thought (CoT) Prompting

In the paper "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models", Wei et al. [85] introduced an innovative prompting technique designed to improve LLM performance on complex reasoning tasks. The core concept of CoT prompting is to make the model's reasoning process explicit. By breaking down tasks into a series of logical steps, each prompt builds upon the previous one, creating a coherent chain of reasoning. This structured approach enables the model to generate more logical, well-organized, and thoughtful responses.

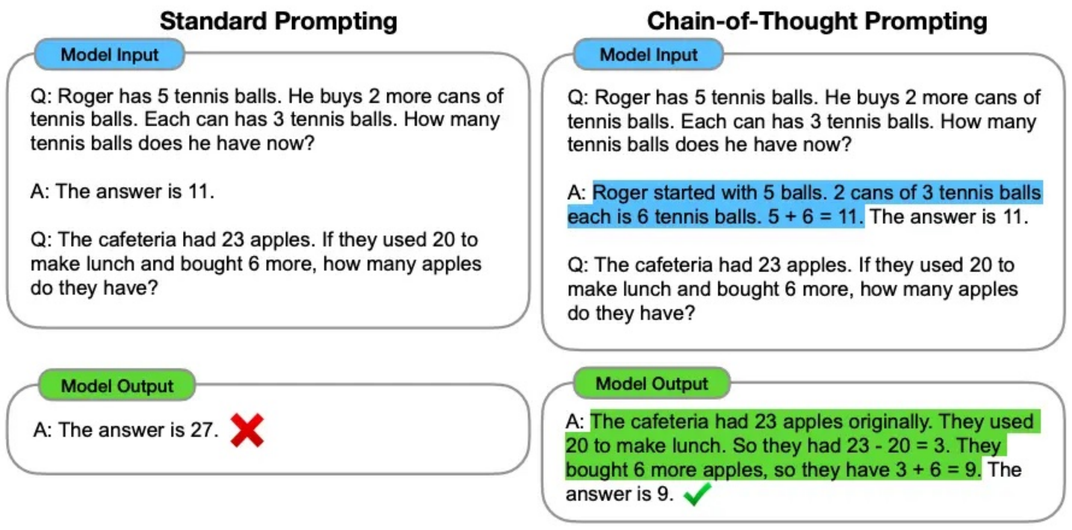

Standard Prompting
Model Input
Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?
A: The answer is 11.
Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?
Model Output
A: The answer is 27.
Chain-of-Thought Prompting
Model Input
Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?
A: Roger started with 5 balls. 2 cans of 3 tennis balls each is 6 tennis balls. 5 + 6 = 11. The answer is 11.
Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?
Model Output
A: The cafeteria had 23 apples originally. They used 20 to make lunch. So they had 23 - 20 = 3. They bought 6 more apples, so they have 3 + 6 = 9. The answer is 9.

Figure 2.7: Example of chain of thought prompting in comparison to standard prompting. Chain of thought prompting allows LLMs to tackle more complex tasks. [85]

There are certain scenarios where CoT prompting may not significantly enhance a model's performance. CoT is effective primarily in sufficiently large models, typically those with 10 billion parameters or more. This is because it leverages various emergent abilities, such as semantic understanding, symbol mapping, maintaining coherence, arithmetic skills, and faithfulness [85]. Additionally, since CoT introduces intermediate reasoning steps into the prompt, it is particularly well-suited for tasks that demand step-by-step reasoning, such as arithmetic, commonsense reasoning, and symbolic problem-solving.

Since the manual creation of high-quality CoT examples is time consuming, Zhang et. al. [97] introduced *Automatic Chain-of-Thought (Auto-CoT) Prompting* that instructs LLMs to generate reasoning chains. Auto-CoT "samples various questions and generates multiple distinct reasoning chains for each, forming a final set of demonstrations." [69]

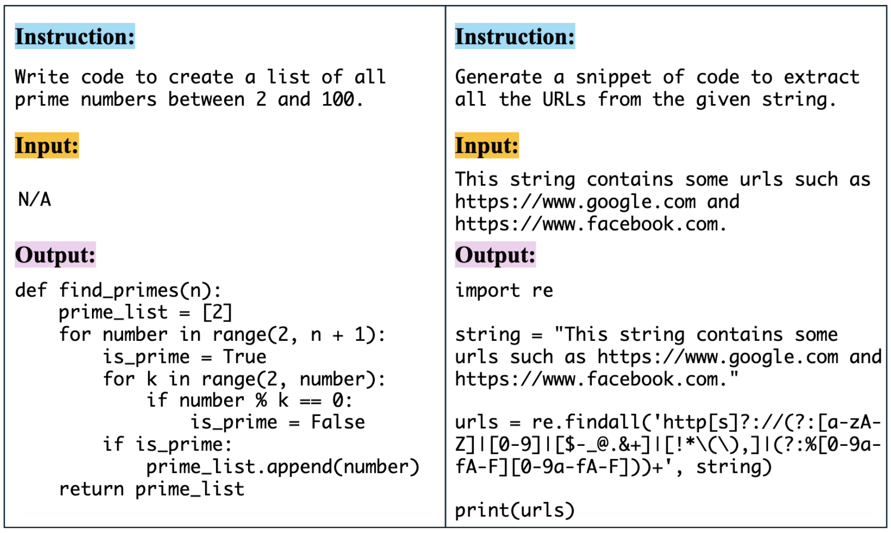
Despite the efficacy of CoT prompting in natural language generation, it demonstrates lower accuracy for code generation tasks [48]. To overcome this weakness Li. et. al. introduced Structured Chain-of-Thought (SCoT), a novel prompting technique for code generation. SCoT prompting improves LLMs' ability to generate structured source code by integrating program constructs such as sequences, branches, and loops into the reasoning process. This method explicitly directs LLMs to approach tasks with a source code-oriented perspective, ensuring better alignment with coding requirements.

2.6. Fine-Tuning Techniques

After pre-training, LLMs exhibit strong performance across a wide range of tasks, including code generation. However, research has shown that their performance can be significantly enhanced through fine-tuning. This process involves adapting LLMs to specific tasks using task-specific data, enabling them to acquire contextually relevant knowledge and perform these tasks with greater accuracy and efficiency. In this section, we will explore the most prominent methods for fine-tuning LLMs for specific downstream tasks.

2.6.1. Instruction Tuning

Instruction tuning refers to the supervised fine-tuning of pre-trained LLMs using a collection of formatted examples in the form of natural language [86]. One of the benefits of instruction tuning is that it equips generic LLMs into domain-specific LLMs. For instance, Flan-PaLM [14] was instruction tuned using medical datasets to create Med-PaLM [76], which now achieves performance levels comparable to those of expert clinicians. To perform instruction tuning on LLMs, it is necessary to first gather instruction-formatted examples. Several hand-crafted instruction tuning datasets have been developed to fine-tune LLMs. However, this process is highly time-consuming, and such datasets are often smaller in size and less diverse due to being manually created. Therefore, several studies such as self-instruct [83] have been proposed to use LLMs to generate instruction tuning datasets. Figure 2.8 demonstrates two exemplars of instruction data sampled from CodeAlpaca [10].



Instruction:

```
Write code to create a list of all
prime numbers between 2 and 100.
```

Input:

```
N/A
```

Output:

```
def find_primes(n):
    prime_list = [2]
    for number in range(2, n + 1):
        is_prime = True
        for k in range(2, number):
            if number % k == 0:
                is_prime = False
        if is_prime:
            prime_list.append(number)
    return prime_list
```

Instruction:

```
Generate a snippet of code to extract
all the URLs from the given string.
```

Input:

```
This string contains some urls such as
https://www.google.com and
https://www.facebook.com.
```

Output:

```
import re

string = "This string contains some
urls such as https://www.google.com and
https://www.facebook.com."

urls = re.findall('http[s]?://(?:[a-zA-
Z]|[0-9]|[$-_@.&+]|[!*\(\),]|(?:%[0-9a-
fA-F][0-9a-fA-F]))+', string)

print(urls)
```

Figure 2.8: Example of instruction data which used to train CodeAlpaca [10].

In the field of SE, recent studies have increasingly adopted LLM-generated instruction methodologies. For instance, WizardCoder [59] utilized Evol-Instruct [89] to fine-tune StarCoder, achieving performance surpassing several models, including Claude-Plus and Bard. Evol-Instruct leverages LLMs to transform given instructions into a set of more complex instructions, enhancing model capabilities. Similarly, Magicoder [87] introduced OSS-Instruct, a method for generating new coding problems from existing open-source code snippets.

LLMs can be fine-tuned using instruction data through two main approaches: full parameter fine-tuning (FFT) and parameter-efficient fine-tuning (PEFT), as described in the two sections below.

Full Parameter Fine-Tuning (FFT)

Full parameter fine-tuning (FFT) involves updating all parameters of a pre-trained LLM. This approach is computationally intensive and requires substantial training data and resources, making it costly. FFT is often preferred in scenarios where computational constraints are less restrictive, as it typically achieves the highest performance. Various LLMs have been fully instruction-tuned for code generation. For example, the fine-tuning of CodeT5+ resulted in InstructCodeT5+, which demonstrated enhanced capabilities in generating code. Similarly, models like WizardCoder, MagiCoder, and others have been fine-tuned to improve instruction-following performance specifically within the domain of code generation.

Parameter-Efficient Fine-Tuning (PEFT)

To address the high computational resource demands of full parameter fine-tuning, Houlsby et al. proposed parameter-efficient fine-tuning (PEFT) [29]. Instead of updating all model parameters, PEFT focuses on a minimal yet impactful subset of parameters, significantly reducing resource consumption. Weyssow et al. [88] explored the use of PEFT for code-generating LLMs, highlighting its effectiveness in efficiently adapting LLMs to task-specific data while maintaining reasonable resource usage. A wide range of PEFT techniques have since been developed, including LoRA [30], prompt tuning [45], and prefix tuning [50].

LoRA proposed by Hu et. al. [30] freezes the weights of the initial model and then injects low-rank trainable matrices into the attention layers of the Transformer architecture. This significantly reduces the number of trainable parameters in the model making the fine-tuning process a lot more efficient. Prompt tuning [45] involves adding virtual tokens at the beginning of the input, which serve as placeholders that provide additional context or instructions to guide the LLM's behavior. In contrast, prefix tuning [50] inserts virtual tokens into every layer of the model, requiring the model to learn additional parameters. These virtual tokens are differentiable, meaning they can be adjusted through back propagation during fine-tuning, while the rest of the LLM remains unchanged. Wang et. al. [84] fine-tuned CodeT5+ for different downstream tasks such as code completion, generation, and search.

3 Methodology

This chapter describes the methodology employed to achieve the research objectives and address the research questions presented in chapter 1. The research followed a structured approach, starting with an examination of the ASML metrology domain and the leveling department within the metrology cluster (See 3.1). This was subsequently followed by the development of a benchmark dataset, which was used to investigate the three research questions (See 3.2). After completing the benchmark dataset, the experimental setup for conducting the experiments was planned, as outlined in 3.3. The results from the experiments were evaluated using the evaluation strategy, as described in 3.4. The research completed with a thorough analysis and summarization of the collected results, providing an answer to the research questions.

3.1. Phase 1: Understanding the ASML Metrology Domain

This study was conducted at the leveling department within the ASML metrology cluster. In this section we introduce what ASML does as a company, what the main function of the leveling department is and the software development process followed within the department.

3.1.1. ASML

Semiconductors and microchips are the backbone of modern devices, from cars and smartphones to MRI scanners and large data centers. The global semiconductor industry has experienced significant growth, with the Semiconductor Industry Association (SIA) reporting total sales of $167.7 billion in the first quarter of 2025 [65]. This figure is anticipated to continue its upward trend in the coming years, according to industry forecasts and statistics from the World Semiconductor Trade Statistics (WSTS) [1].

The production of semiconductors involves a complex, multi-step process, with photolithography playing a crucial role. This critical stage involves scanning the wafer with high precision and then printing the chip's pattern onto semiconductor wafers [40]. Figure 3.1 shows a simplified version of the life of a wafer during the photolithography stage. ASML, is one of the leading manufacturers of such lithography machines, that enable chip manufacturers to achieve nano-meter level precision.

Initially, ASML's machines scanned and printed in sequence, but the introduction of TWINSCAN machines enabled parallel processing, boosting chip manufacturing productivity. The TWINSCAN machines, are still the only lithography systems with two wafer table modules. "When the wafer on table one is being exposed, another wafer is loaded on table two and then aligned and mapped" [81].

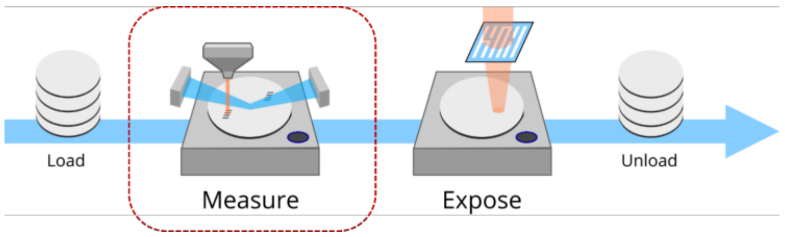


Figure 3.1: Simplified view of the life of a wafer in the scanner. The wafer is loaded, measured, exposed, then unloaded [7].

As the semiconductor industry continues to push the boundaries of precision and accuracy, the importance of precise measurement and control cannot be overstated. This is where metrology comes into play.

3.1.2. Metrology

In general terms, metrology refers to the science of measurement, encompassing both theoretical and practical aspects [8]. Within the lithography machines, the primary objective of metrology is to provide models, functions and system software and use them to determine the set points for all relevant subsystems during exposure. This in turn guarantees the quality and performance of the semiconductor products. To achieve this, metrology employs a range of measurement methods and equipment to collect data and inspect various process parameters, such as alignment position, as well as other wafer properties [60]. Within metrology, the task of *leveling* involves measuring the vertical position of the wafer on the chuck using level sensors. This information is utilized to maintain the wafer in focus during the exposure stage, thereby ensuring optimal accuracy and precision during this critical process. The measuring process can also be performed prior to exposure, in which case a level map is executed when imaging the actual dies.

3.1.3. DCA Architecture

The ASML leveling department maintains an extensive code base, allowing them to process large amounts of data collected by the leveling sensors and perform computations on them. The 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 concern. To model the behavior of its machines, ASML uses the ASML Software Modeling Environment (ASOME) modeling tool which provides tools for model analysis and automated construction of software systems.

By applying the DCA architecture, data can be stored and manipulated separately from the environment where the behavior is defined. Similarly, algorithms can be designed independently using the stored data. Additionally, the DCA architecture allows for formal (mathematical) verification, enabling the control elements of the software to be mathematically proven to meet the specified requirements. To ensure efficient verification, it is essential to keep the state space small, which is achieved by specifying control behavior in terms of state machines and minimizing the influence of data on decision logic. By doing so, the DCA architecture enables the verification of control behavior using formal methods and tooling, without being hindered by the complexity of data or algorithms. This approach allows for more efficient and effective verification of the system's design [3].

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 [77]. 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 [77]. 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 [77].

An overview of the DCA architecture is shown in figure 3.2 below.

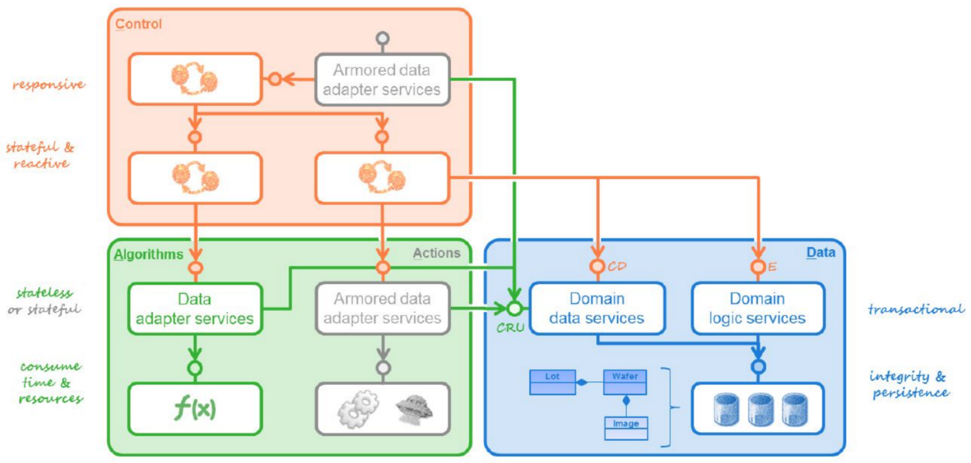


Figure 3.2: DCA architecture used in the ASML metrology software separating the data, control and algorithm components [44].

Data Component

Data elements from an ASML component are represented using one or more domain models that consist of the following elements:

- **Domain Interface:** *Domain interfaces* allow for dependencies between different models, allowing model elements to be shared across different models. These interfaces can contain various model elements such as, but not limited to enumerations, entities, and value objects.
- **Value Objects:** Algorithms are often divided into several steps with each their own in- and outputs. This results in temporary data objects in between the different steps. *Value objects* are used to store and manage the intermediate results from these algorithms. They can contain attributes and associations to other entities or value objects.
- **Entities:** *Entities* are model elements that contain value objects, attributes and associations to other entities.
- **Enumerations:** *Enumerations* contain a collection of constants that are known as 'enumeration literals'.

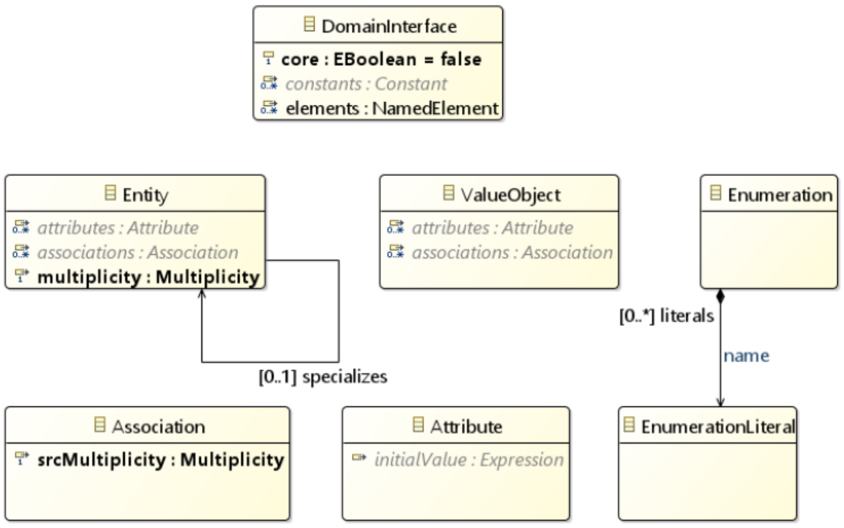


Figure 3.3: Basic elements present in a domain model [77].

Upon defining the data model, ASOME automatically generates various code components that conform to the specified model.

All persistent data is stored in a data repository that can be accessed through RepoAccess and Navigator classes. *RepoAccess* objects are used to store, retrieve and update entities of a specific repository. Such an abstraction is used to limit the scope of what domain services can do with a repository. The *Navigators* are used navigate through a domain model offering its clients a way to reach from A to B without explicitly knowing the path between them. Navigators use 'Routers' that provide single hop movement between Entities. A hop in a router returns the router of the next entity. After a chain of hops we can call the `getEntity()` or `getId()` method to retrieve the entity itself.

To enable the algorithm and control components to interact with the data component, the data component exposes an interface, allowing these components to access and interact with it (see figure 3.4). It consists of the *domain logical services*, which are provided to the control component and *domain data services*, which is provided to both the control and algorithm (durative actions) components. The sections below describe the two domain services in detail.

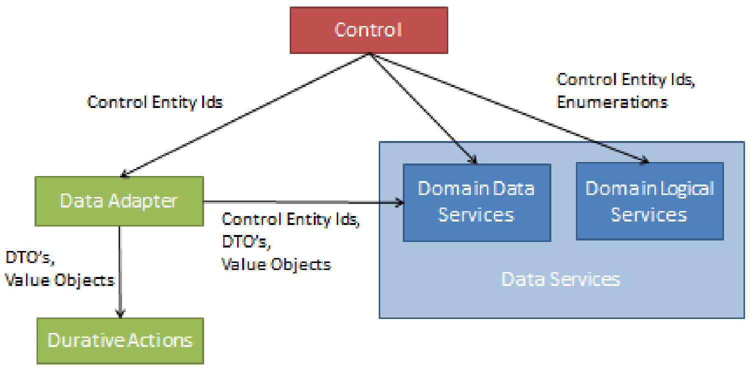


Figure 3.4: Domain services that are exposed to the control and algorithm component to be able to access information from the data component

Domain Logical Service The domain logical services provide a *predicate* interface to the control component. This allows the control component to make decisions based on the data. The predicates implementation simply checks the existing data and returns an enumerable result, such as a boolean, enumeration, or integer value. If the provided entry point is not the target entity, a navigator may be necessary to route to the target entity and verify the existing data. Figure 3.5 provides an overview of the predicate interface that is made accessible to the control component. Listing 3.1 defines a boiler plate for defining a predicate.

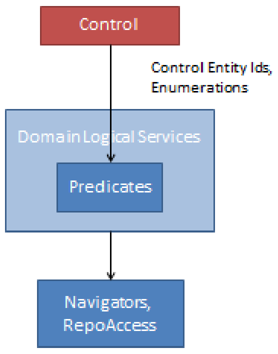


Figure 3.5: Diagram of the domain logical services that provide a predicate interface to the control component.

```
virtual bool isDataAPresent(TypeBEntity bId) const
{
    return !domainNavigator->getAllTypeAEntities(bId)->empty();
}
```

Listing 3.1: Template code for defining a predicate interface to the control component

Domain Data Services The domain data services provide two main kinds of interfaces. To the control component it provides *lifecycle* interfaces and to the the algorithm component it provides the *garage* interface.

The *lifecycle* interface manages the creation and destruction of control entities. It sets up and makes the entity available for use, and defines how it should be destroyed or deleted when no longer needed, with any related data entities being deleted through cascaded deletion. Essentially, the lifecycle interface acts as a manager for the control entity's entire lifespan, from creation to destruction, ensuring it is properly set up and torn down, and handling related data accordingly.

The *garage* interface is used to store and retrieve data to/from a data repository. To be able to perform operations, garages need to use additional modules such as Navigators, RepoAccess and EntityFactories. Garages do not operate directly on data entities but make use of Data Transfer Objects (DTOs) instead, thus they often need te make conversions between entities and DTOs. A data transfer object [32] is an object that carries information between processes, it is used when we do not want to expose all internal domain details out to the consumers of the data. Figure 3.6 provides an overview of the interactions that the domain data services has with the algorithm and control components.

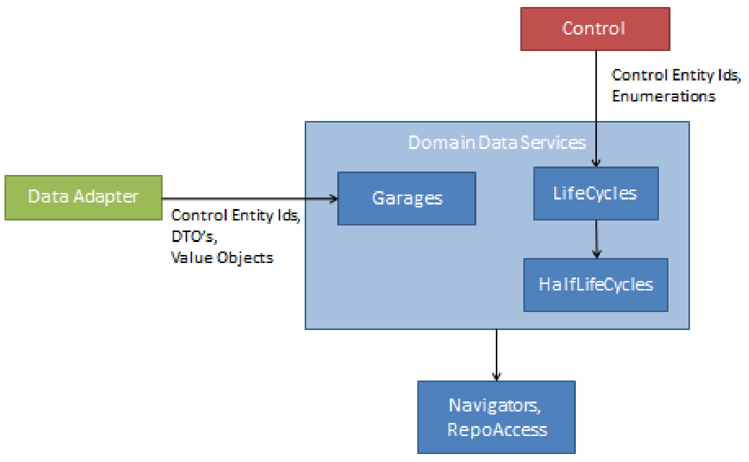


Figure 3.6: Diagram of the domain data services that provide the lifecycle and garage interface.

Garages are only used for the storage and retrieval of data but they do not offer the possibility to delete data. Deletion of data happens by cascaded deletion when the control entity it refers to gets deleted by its lifecycle. The implementation for storing and retrieving data follows a fairly standard pattern and is therefore a perfect candidate to be generated automatically using LLMs. There are three main types of garages: a store-garage that only has storage capabilities, a retrieve-garage that only has retrieval capabilities and a store-retrieve garage that performs both operations.

To retrieve data from an entity, a DTO object needs to be created out of the entity and provided to the client. To create a DTO getter for an algorithm two steps are needed:

1. Navigate from the provided entity or entityID to the requested one
2. Provide a DTO object out of the requested EntityType. To perform this task there is a generated auxiliary class that provides entity to DTO transformations, the AdapterDTO.

Listing 3.2 defines the boiler plate code for retrieving an entity within a garage.

```
virtual TypeADTOPtr getTypeA(const TypeBEntityId& bId)
{
    TypeAPtr aEntity = domainNavigator->findTypeA(bId);
    return boost::make_shared<ConstTypeAdapterDTO>(aEntity);
}
```

Listing 3.2: Template code for a retrieve garage that retrieves data from an entity.

To store data into an entity, two main things are needed: the data that needs to be stored provided in the form of a DTO, and the association to the entity. Sometimes we may have access to the direct dependency but occasionally we only have access to an indirect dependency in which case a Navigator is used to reach the direct dependency. For the creation of an entity we make use of an EntityFactory and to store the entity into a repository we use a RepoAccess object. Listing 3.3 defines the boiler plate code for storing data into an entity.

```
virtual void storeTypeA(TypeADTOPtr& aDTO, typeCEntityId& cId)
{
    TypeBEntityId bId = domainNavigator->findBId(cID);
    aEntity = aFactory->create(
        aDTO->value1, aDTO->value2,
        bId);
    aRepo->store(aEntity);
}
```

Listing 3.3: Template code for a store garage that stores data into an entity

While ASOME also facilitates the specification of control and algorithm models, these were not considered for the purpose of this study. The focus of this study was purely focussed on the Data component of ASOME.

3.2. Phase 2: Benchmark Dataset Creation

For the purpose of this study it was chosen to focus on code generation of garages within the leveling repository. This choice was motivated by the fact that garages typically have a standardized structure, yet contain small nuances that are difficult to capture using traditional scripting approaches, making them an ideal test case for evaluating the code generation capabilities of LLMs within ASML. The remainder of this section describes the benchmark creation, which involves the collection of code implementation, context-files, unit-tests and other meta-data related to the garage.

The majority of garages in the leveling repository adhere to a consistent naming convention. This naming convention was used as a search criterion to identify all garages in the repository, yielding a total of *168* garages. However, upon closer inspection, it was found that six garages shared the same name but had different functional logic. To avoid confusion and complications in finding the correct context files, these garages were excluded from the benchmark dataset. As a result, the final benchmark dataset consisted of *156* garages. The file path and implementation of these garages were stored in the benchmark dataset.

Unit Tests To evaluate the functional correctness of the generated code, the unit tests associated with each garage were also collected. Unfortunately, not all garages had accompanying unit tests. Of the 156 collected garages, only 42 garages had associated unit tests, representing approximately 27% of the total dataset.

Context Files To provide the LLM with more information about the garage, we collected all files related to it as contextual input. A file is considered relevant if it is directly or indirectly referenced through import statements in the garage or any of its dependencies determine recursively. The related files are also assigned a ranking or depth value, indicating their distance from the garage in the import tree. The depth value is determined by the file's position in the import hierarchy: a file directly imported by the garage is assigned a depth of *1*, while a file imported by one of the garage's imported files is assigned a depth of *2*, and so on.

Although the method for gathering related files for a garage was comprehensive and sound, it sometimes collected too many files, some of which were not essential. To refine the collection, a distinction was made between related files automatically generated by ASOME and those written by humans. It was decided to exclude auto-generated files with a rank or depth greater than 2 from the benchmark dataset. This significantly reduced the number of collected related files, ensuring that only the most important files were included in the benchmark.

It was also observed that many of the context files generated by ASOME were very long, often exceeding 2000 lines of code. Due to the limited context window of many LLMs, it was decided to reduce the size of these context files. To achieve this, an LLM was used to summarize the contents of each file, and the resulting summary was used in the benchmark instead of the original code implementation. The model employed for summarization was *Qwen2.5-Coder-32B-Instruct* [31] with Q8_0 quantization and a context length of 32k. A total of 12 file types were selected for summarization by an LLM.

Beyond summarization, we gathered additional metadata about the context files. We computed embeddings for all context files using the *BGE-M3* model [11] with an F32 precision level and an 8K token context size. These embeddings were used to prioritize context files during the selection process, determining which files to include in the prompt. Additionally, we calculated the token counts for each context file across the three model families utilized in our study: Qwen2.5, DeepSeek, and Gemma. This token length analysis enabled us to identify which files would fit within our context window and which would exceed it.

The overall format of the benchmark dataset is shown below in listing 3.4:

```json
[
    {
        "name": <Garage_Name>,
        "path": <Garage_FilePath>,
        "component": <Garage_Component>,
        "solution_code": <Garage_Implementation>,
        "related_files": [
            {
                "file_path": <ContextFile_FilePath>,
                "depth": <ContextFile_Depth>,
                "implementation": <ContextFile_Implementation>,
                "embedding": <ContextFile_Embedding>,
                "Qwen2.5-Tokens": <ContextFile_QwenTokens>,
                "DeepSeek-Tokens": <ContextFile_LlamaTokens>,
                "Gemma-Tokens": <ContextFile_GemmaTokens>
            }, ... , {}
        ],
        "test_directory": <Garage_TestDirectory>,
        "test_file_name": <Garage_TestName>
    }, ... , {}
]
```

Listing 3.4: JSON skeleton template of the benchmark dataset created for the experiments.

3.2.1. Benchmark Analysis

The collected garages for the benchmark dataset exhibit a wide range of sizes. Garage size can be quantified by the number of lines of code written for each garage. Figure 3.7 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.

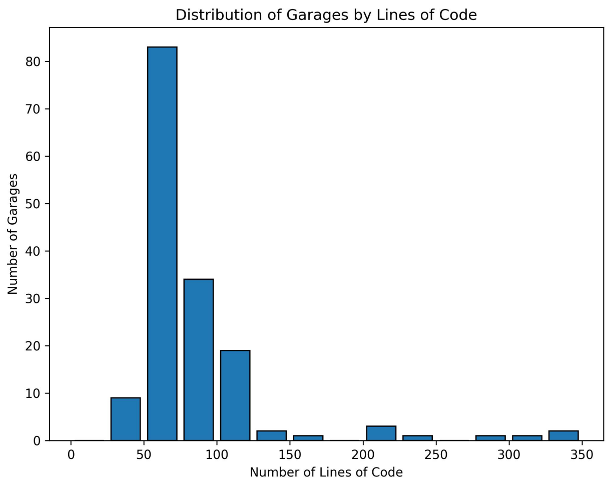


Figure 3.7: Histogram with the distribution of garages in the benchmark by the number of lines of code.

Another insight about the benchmark dataset is the number of context files associated with each garage. Figure 3.8 illustrates the distribution of the number of context files per garage and their corresponding frequencies. As shown in the figure, the majority of garages have fewer than 15 context files associated to it, while a very small fraction of garages have more than 15 context files.

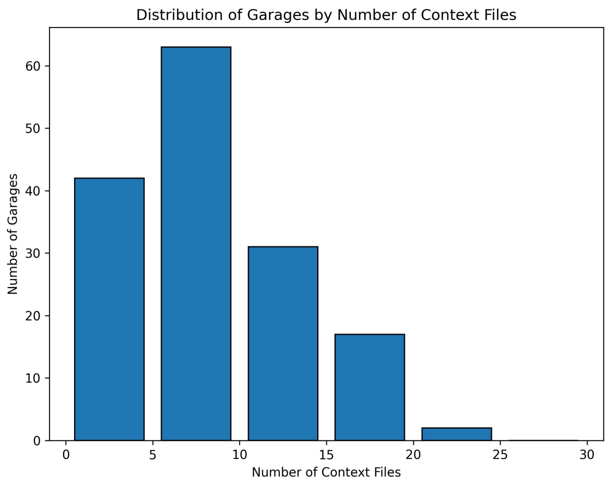


Figure 3.8: Histogram with the distribution of garages in the benchmark by the number of associated context files.

It can be hypothesized that as the complexity of a garage increases (as reflected by a higher number of lines of code), the number of context files required to implement it also tends to increase. Figure 3.9 presents a scatter plot of the relationship between the number of context files and lines of code, accompanied by a red regression line. The regression line suggests a low positive correlation between the two variables, although the correlation is not strong. Furthermore, the *pearson correlation coefficient* ($r = 0.31$) confirms that the positive correlation is very low, almost negligible. The hypothesis that we had about the two parameters is therefore not supported.

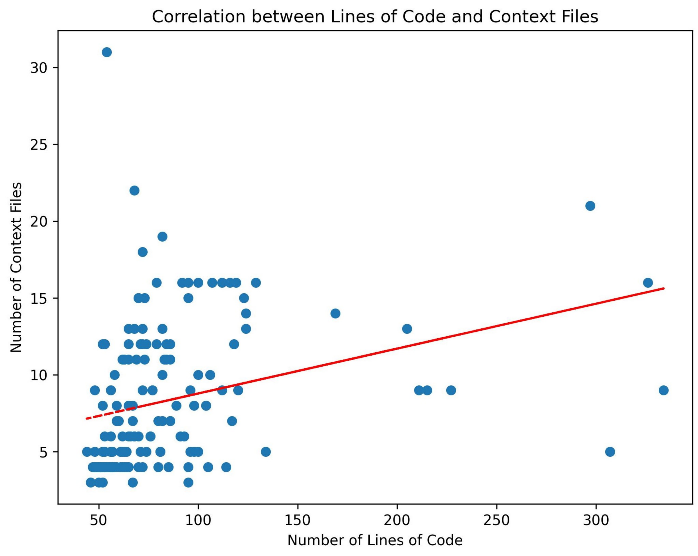


Figure 3.9: Scatter plot with the correlation between the number of lines of code in a garage and the number of context files associated to it.

3.3. Phase 3: Experimental Setup

The following section describes the experimental setup and methodology used to answer the three research questions, providing a clear and concise overview of the study's design and implementation. The results of each experiment were evaluated according to the procedure detailed in Section 3.4, ensuring a standardized assessment of the findings.

3.3.1. Experiment 1 (RQ1)

The first experiment examined the influence of prompting techniques on generated code quality. Five distinct prompting techniques, previously described in Section 2.5, were selected for this study and are listed below.

Prompting Techniques
Zero-Shot Prompting
One-Shot Prompting
Few-Shot Prompting
One-Shot Chain-of-Thought Prompting
Few-Shot Chain-of-Thought Prompting

Table 3.1: List of prompting techniques used for experiment 1.

The specific prompts used for these experiments have been shown in Appendix A

Due to various security related restrictions it was decided to work with LLMs that were already accessible within the ASML eco-system. There were three main models that were available to use at the time of experiment 1 as shown in table 3.2 below.

Model	Precision Level	Context Window
meta-llama/Llama-3.1-8B-Instruct	Q8_0	32k tokens
meta-llama/Llama-3.3-70B-Instruct	Q8_0	16k tokens
Qwen/Qwen2.5-Coder-32B-Instruct	Q8_0	32k tokens

Table 3.2: Large language models that were available to use for experiment 1 within the ASML infrastructure.

Out of the three available models, we selected the Qwen2.5-Coder model due to several key advantages it offered over the other options. Notably, the Qwen2.5-Coder model had a larger context window compared to the Llama 3.3-70B model, allowing it to process and understand more extensive code sequences. Additionally, Qwen2.5-Coder demonstrated higher performance on the BigCodeBench benchmark [101], indicating its superior capability in handling complex coding tasks. Furthermore, as a code-specific model, Qwen2.5-Coder was more suitable to our experimental needs, making it the most logical choice for our investigation.

We opted to utilize the default sampling parameters, as fine-tuning these parameters fell outside the scope of this experiment. Specifically, the temperature was set to 0.7, the top-p parameter to 0.8, and the top-k parameter to 20. The temperature of 0.7 indicates a balance between randomness and predictability, allowing for some diversity in the output while still maintaining coherence. The top-p value of 0.8 means that the model considers the top 80% of the most likely next tokens, which can help in generating relevant and contextually appropriate text. The top-k value of 20 further refines this by limiting the options to the 20 most probable tokens, potentially enhancing the output's coherence and reducing the likelihood of hallucination.

The Qwen2.5-Coder model's context window is limited to 32k tokens, necessitating a prioritization strategy for including files in the context. To determine the optimal allocation, we first identified the largest prompt used, which was the *Few-Shot Chain-of-Thought* prompt, taking up approximately 3254 tokens. Additionally, we considered the largest garage in our dataset, which takes up 3418 tokens, to ensure sufficient space for output tokens. This resulted in a minimum requirement of 6672 tokens to accommodate both the prompt and output. To be conservative, we allocated 25k tokens for context files, leaving 7k tokens for the prompt and output, thereby ensuring sufficient space for the model's operations.

The prioritization of context files was based on a two-stage approach. Firstly, files were prioritized according to their depth level, with files having a lower depth value taking precedence over those with higher depth values. In cases where not all files from the same depth level could be included due to token limitations, a secondary prioritization was applied based on the cosine similarity between the context file and the prompt. This involved selecting files that exhibited a higher degree of similarity to the prompt, thereby maximizing the relevance of the included context. Notably, this similarity-based approach is also commonly employed by Retrieval-Augmented Generator (RAG) based LLM, highlighting its effectiveness in optimizing context selection.

Code generation for this experiment was performed on an NVIDIA A100 Tensor Core GPU, featuring 80GB of VRAM, which offered a high-performance computing environment with enough memory resources.

3.3.2. Experiment 2 (RQ2)

The second experiment investigated the performance difference between generic LLMs and code-specific LLMs in generating garages within ASML. To ensure a fair comparison, we utilized code-specific models fine-tuned from generic LLMs, rather than foundational code-specific LLMs, as this allowed us to compare models originating from the same foundation and having the same architecture. The models used in this experiment are presented in Table 3.3 below. All models were used in their default configuration using their default sampling parameter values.

Family	Model	Precision	Cont. Window	Model Size
Qwen2.5	Qwen2.5-7B-Instruct	Q8_0	32k tokens	7B params
	Qwen2.5-Coder-7B-Instruct	Q8_0	32k tokens	7B params
Gemma	gemma-7b-it	Q8_0	8k tokens	7B params
	codegemma-7b-it	Q8_0	8k tokens	7B params
Deepseek	DeepSeek-V2-Lite-Chat	Q8_0	32k tokens	16B params
	DeepSeek-Coder-V2-Lite-Instruct	Q8_0	32k tokens	16B params

Table 3.3: Large language models used for experiment 2 and the corresponding precision, context window, and model size.

In all experiments, single-shot CoT prompting was employed as the prompting technique. This approach was chosen due to its proven strength in research for code generation tasks [93], as well as its efficiency in terms of token usage, requiring fewer tokens compared to few-shot CoT prompting. The prompt is provided in appendix A

The selection of context files was done using the same methodology as described in 3.3.1, for the Qwen2.5 and DeepSeek models, which had a context window of 32k tokens. In contrast, the Gemma models had a lower maximum context window, so the allocated space for context files was limited to 4k tokens. The prioritization of context files followed the same procedure as in the first experiment. even though, the context window for the Gemma models is significantly smaller than that of the other two models, it is sufficient for the purpose of this experiment, as the goal is to investigate the difference between generic and code-specific models, rather than comparing the models directly with each other.

Code generation for the Qwen2.5 and Gemma models was done using a NVIDIA T4 Tensor Core GPU, equipped with 16GB of VRAM. Since the DeepSeek models were larger in size, inference was performed on an NVIDIA A100 Tensor Core GPU, featuring 80GB of VRAM.

3.3.3. Experiment 3 (RQ3)

The third and final experiment in this study 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. Table 3.4 below presents the different model sizes used in this experiment.

Model
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

Table 3.4: Model sizes used for experiment 3.

In experiment three, all models used a consistent set of configurations to ensure uniformity and comparability of results. The configurations consisted of Q8_0 quantization, a 32K context window, a temperature of 0.7, top-p of 0.8, and top-K of 20. The token allocation for context files and prompt/output was 25k and 7k, respectively. The methodology followed for prioritizing the context files was the same as described in 3.3.1.

Single-shot CoT prompting was used as the prompting technique, given its proven effectiveness in previous research [93]. The prompts used were identical to those used previously, as listed in appendix A. This consistent approach allowed for a reliable comparison of the models' performance in experiment 3.

The code generation of the 14B and 32B models were done using the NVIDIA A100 Tensor Core GPU, while the generation of the remaining models were done using the NVIDIA T4 Tensor Core GPU.

3.4. Phase 4: Evaluation Strategy

This section outlines a systematic approach to assess the performance and effectiveness of code generated by the LLMs. The primary objective of this evaluation is to collect evidence that addresses the research questions posed in the introduction (Section 1).

To assess the LLMs in a realistic setting, no post-processing was applied to the generated output. As a result, the output may occasionally include extra text beyond the requested code. This was intentionally not filtered out to simulate an ideal scenario where the generated code is directly placed in the codebase without any additional processing.

3.4.1. Match-based Metrics

At the first stage of evaluating the performance of the generated code, we employed various code-similarity metrics. The metrics used in this study for comparing the generated output and the reference implementation are BLEU, CodeBLEU and ROUGE. The BLEU metric is as described in section 2.4.1. We use BLEU-4, which compares the n-gram overlap between the candidate and reference text, using up to 4-grams. The ROUGE metrics come in three main variants: ROUGE-1, which measures the overlap of uni-grams (individual words) between the generated and reference code; ROUGE-2, which evaluates the overlap of bi-grams (pairs of words); and ROUGE-L, which assesses the longest common subsequence between the two, providing a measure of the similarity in code structure and sequence. Finally, we also used CodeBLEU a code-specific similarity metric which combines four components: n-gram match, weighted n-gram match, syntax match, and data-flow match. Each component was weighed equally at 0.25, similar to the original paper [67] in which it was introduced. This allows to capture both lexical and semantic similarity in a balanced manner.

3.4.2. 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. This stage aimed to provide a more comprehensive understanding of the generated code's functionality. The three types of execution-based metrics used in this evaluation are described below.

Buildability

The first execution-based metric we assessed for the generated code was its capacity to produce output that enables successful project and repository builds. While traditional execution-based metrics, as noted in a study by Chen et al. [12], typically focus on file- or function-level evaluations, they often neglect repository-level analysis. For example, a model may generate syntactically correct functions that fail to integrate with the broader project due to unresolved dependencies or data flow mismatches. Consequently, evaluating the buildability of the generated code was deemed essential. If the generated code hinders the project's build process, it becomes effectively useless to the user.

To verify the integrability of the generated code within the project, we substitute the original file in the repository with the generated code file and then attempt to build the entire project. This approach ensures that the generated code is valid within the project's broader context, and that all dependencies can be successfully resolved. By taking this step, we can determine whether the generated code is compatible with the existing project structure and dependencies, and identify any potential issues that may arise during integration. To ensure a consistent evaluation environment, we use a specific commit from the project, which allows us to isolate the impact of the generated code file. By keeping all other files in the project unchanged, except for the generated file being evaluated, we can accurately assess its ability to be be placed in the repository.

To the best of our knowledge, there is no existing metric that assesses the buildability of generated code. Consequently, we propose a novel performance metric, termed *build@k*, which draws inspiration from the pass@k metric [42].

The *build@k* metric, takes *k* code samples per garage, a garage is considered as buildable if it successfully integrates into the project and passes the build pipelines, the total fraction of buildable garages is reported. This metric provides a quantitative evaluation of the generated code's buildability, enabling a more comprehensive assessment of its practical usefulness and effectiveness within the context of the larger project.

By introducing the *build@k* metric, we aim to address the limitations of existing metrics, which often focus solely on syntactic correctness or functional accuracy, and neglect the crucial aspect of buildability. The *build@k* metric offers a more global evaluation of generated code, taking into account its ability to integrate with the project's dependencies and build pipelines, thereby providing a more accurate measure of its overall quality and usability.

Formula 3.1 as given below formally defines the *build@k* metric.

$$\text{build@k} = \frac{1}{N}\sum_{i=1}^{N}\begin{cases}1 & \text{if any of } k \text{ generated code instances for garage}_i \text{ is successfully built}\\ 0 & \text{otherwise}\end{cases} \tag{3.1}$$

Unit Tests

The second execution-based metric used to evaluate the generated code involves assessing the execution of unit tests associated with the generated code. To evaluate a generated piece of code against unit tests, two primary criteria must be met:

- The garage for which the code is being generated must have existing unit tests (only 27% of garages have unit tests)
- The generated code must be buildable within the project.

For this evaluation, we employ the pass@k metric, as proposed by Kulal et al. [42]. This metric involves generating *k* code samples per garage, where a garage is considered solved if at least one sample passes the unit tests. The pass@k metric then reports the total fraction of garages that are successfully solved. We only consider garages with existing unit tests for this metric, as it would be unfair to penalize garages that lack unit tests.

The pass@k formula we use can be formally defined as follows (see 3.2)

$$\text{pass@k} = \frac{1}{N}\sum_{i=1}^{N}\begin{cases}1 & \text{if any of } k \text{ generated code instances for garage}_i \text{ passes all unit tests}\\ 0 & \text{otherwise}\end{cases} \tag{3.2}$$

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 [12]. While most existing metrics focus solely on assessing the functional correctness of generated code, the crucial aspect of code quality is often neglected. This oversight is concerning, as code quality has a large impact on the overall usability, reliability, and longevity of the generated code and tools that generate code.

To address this limitation, we have chosen to utilize the TICS tool, an evaluation platform used at ASML [33], 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 category of coding standards comprises a set of rules that are checked for within the submitted code, as shown by the rules outlined in Table 3.5. These rules are 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 [34].

Rule	Category	Short Description
C.21	Classes and class hierarchies	If you define or =delete any copy, move, or destructor function, define or =delete them all
F.54	Functions	When writing a lambda that captures this or any class data member, don't use [=] default capture
INT#027	Class Interface	If you override one of the base class's virtual functions, then you shall use the "override" or "final" keyword
CFL#001	Control Flow	Statements following a case label shall be terminated by a statement that exits the switch statement
NAM#002	Naming	Do not use identifiers which begin with an underscore ('_') followed by a capital

Table 3.5: Example rules in the Philips C++ Coding Standard used in the TICS tool [34].

To evaluate the code quality of the generated code, we will analyze six key metrics using TICS:

1. The total number of violations across all severity levels in the buildable garages.
2. The average number of violations per buildable garage.
3. The net (delta) change in TICS scores, calculated as the difference between the original and generated implementations.
4. The total number of critical violations (levels 1-7) in the buildable garages.
5. The number of critical violations per buildable garage.
6. The total number of non-critical violations (levels 8-10) in the buildable garages.

This analysis will provide a detailed insight into how the code quality of the generated files compares to the human-written implementation. This will allow us to evaluate the generated code's maintainability, readability, and performance efficiency in relation to the human-written code, and gain a deeper understanding of the strengths and weaknesses of the code generation process.

3.4.3. Human Evaluation

To ensure a comprehensive evaluation process, human evaluation is necessary, as it provides robust and reliable results. As the final step in our evaluation strategy, human 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. However, human evaluation also has a notable drawback: the introduction of biases and inconsistencies by human evaluators, which can impact the reliability of the evaluation process.

3.5. Phase 5: Result Analysis

In the final phase of our study, we will analyze the results we have collected and interpret them to answer our research questions. We will examine the data to see how well our code generation approach worked and compare the generated code to human-written code. We will also look for areas where our approach can be improved and identify potential future directions for research. By carefully reviewing the results, we aim to gain a better understanding of what works and what does not, and use this knowledge to inform future developments in using LLMs for code generation in domain-specific environments.

4 Results

This section presents the results from the experiments conducted as described in section 3.3. The evaluation of these experiments followed the strategy outlined in section 3.4. The results presented here provide quantitative measurements to address the research questions posed earlier. This section is divided into three subsections, each of which presents the results for one of the three experiments conducted.

4.1. Experiment 1: Prompting Techniques

The first experiment examined the impact of prompting techniques on the domain specific code generation ability of LLMs. This section presents the results of the experiment, including both match-based and execution-based outcomes, shown in tables and figures. A detailed discussion and interpretation of these findings can be found in Section 5.

4.1.1. Match-Based Metrics

The first aspect that we consider for the generated code is the similarity of the generated code to the reference implementation. The formula used to calculate the mean BLEU, CodeBLEU and ROUGE scores for $k = 1, 3, 5$ is given by 4.1:

$$\text{Mean score: } \frac{1}{N}\sum_{i=1}^{N} \max\{S_{i,1}, S_{i,2}, ..., S_{i,k}\} \tag{4.1}$$

Where $N = 156$ is the total number of tasks, and $S_{i,j}$ represents either the j^{th} BLEU score $B_{i,j}$, the j^{th} CodeBLEU score $CB_{i,j}$ or the j^{th} ROUGE score $R_{i,j}$ for the i^{th} task. For $k = 3$ and $k = 5$ we considered the highest BLEU, CodeBLEU and ROUGE scores from all the generated codes for that task.

Table 4.1 and 4.2 present the mean BLEU, CodeBLEU and ROUGE scores of the generated code across all garages for the different prompting techniques and varying values of k.

	BLEU			**CodeBLEU**		
	k=1	*k=3*	*k=5*	*k=1*	*k=3*	*k=5*
Zero-Shot Prompting	0.449	0.487	0.509	0.320	0.360	0.380
One-Shot Prompting	0.538	0.580	0.598	0.395	0.440	0.456
Few-Shot Prompting	0.620	**0.665**	**0.676**	0.470	0.513	0.531
One-Shot CoT Prompting	0.615	0.652	0.670	0.456	0.502	0.512
Few-Shot CoT Prompting	**0.625**	0.663	0.670	**0.483**	**0.523**	**0.534**

Table 4.1: Mean BLEU and CodeBLEU scores of generated code across five different prompting techniques, with k values of 1, 3, and 5.

	ROUGE-1			ROUGE-2			ROUGE-L		
	k=1	*k=3*	*k=5*	*k=1*	*k=3*	*k=5*	*k=1*	*k=3*	*k=5*
Zero-Shot Prompting	0.624	0.654	0.667	0.449	0.484	0.499	0.509	0.542	0.558
One-Shot Prompting	0.679	0.710	0.724	0.525	0.566	0.581	0.578	0.611	0.625
Few-Shot Prompting	**0.753**	**0.780**	**0.788**	**0.626**	**0.661**	**0.671**	**0.646**	**0.679**	**0.691**
One-Shot CoT Prompting	0.749	0.776	0.787	0.620	0.650	0.663	0.641	0.669	0.682
Few-Shot CoT Prompting	0.751	0.777	0.782	0.621	0.656	0.664	0.641	0.676	0.680

Table 4.2: Mean ROUGE-1, ROUGE-2, and ROUGE-L scores of generated code across five different prompting techniques, with k values of 1, 3, and 5.

The results from tables 4.1 and 4.2 reveal a substantial difference in the similarity of the generated code, produced using different prompting techniques. Notably, zero-shot prompting exhibits the lowest similarity, suggesting its ineffectiveness in generating domain specific code. One-shot prompting also performs relatively poorly, although slightly better than zero-shot prompting. In contrast, few-shot, one-shot CoT, and few-shot CoT prompting techniques demonstrate the highest performance, with minimal differences in their mean BLEU, CodeBLEU and ROUGE scores.

In terms of BLEU score, few-shot CoT prompting achieved the highest mean score at $k = 1$, while few-shot prompting performed marginally better for the remaining k values. Similarly, for the ROUGE scores, few-shot prompting consistently performed the best across all values of k. Finally, in terms of CodeBLEU scores, few-shot CoT prompting showed the best performance for all values of k.

The similarity scores from experiment 1 also show that the similarity between the generated code and the reference code increases as the number of generations (k) increases. This trend is consistent across all prompting techniques and metrics. The most significant improvement in similarity is observed between $k = 1$ and $k = 3$, with a substantial jump in scores across all metrics and prompting techniques. In contrast, the difference in similarity between $k = 3$ and $k = 5$ is less pronounced, suggesting that the rate of improvement may be diminishing as the number of generations increases.

Figures 4.1, 4.2 4.3, 4.4, and 4.5 provide a visual representation of the similarity scores between the generated code and the reference code for different prompting techniques. The bar charts illustrate the performance trend, where zero-shot and one-shot prompting techniques consistently exhibit the lowest similarity scores, while the other techniques demonstrate superior and comparable performance. Moreover, the figures suggest a positive correlation between similarity and the number of generations (k), with similarity scores increasing as k increases. This visualization reinforces the observation that multiple generations may lead to improved code similarity, and that certain prompting techniques outperform others in terms of generating code that closely resembles the reference code.

The error bars in the figures represent the standard deviation of the various similarity metrics across all generations. The relatively small error bars suggest low variance in the values, indicating that most values cluster around the mean.

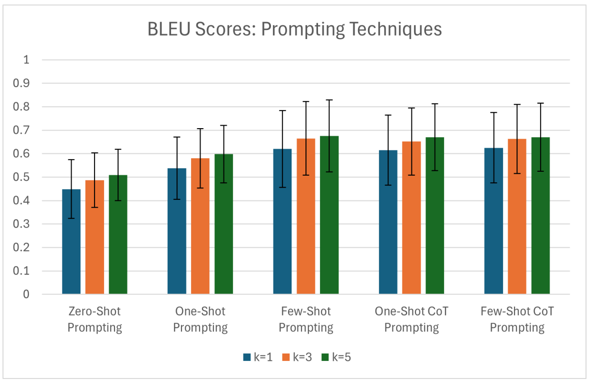


Figure 4.1: Bar chart with mean BLEU score of generated code across five different prompting techniques, with k values of 1, 3, and 5. Error bars represent the standard deviation.

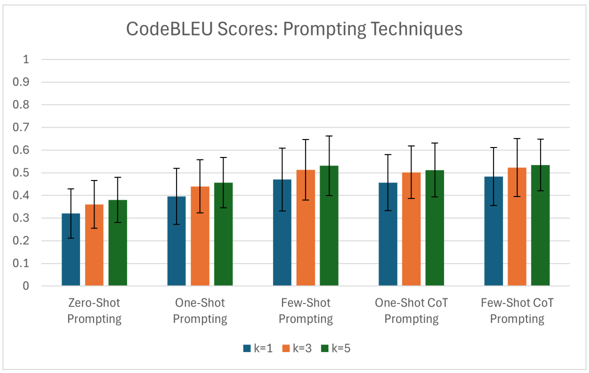


Figure 4.2: Bar chart with mean CodeBLEU score of generated code across five different prompting techniques, with k values of 1, 3, and 5. Error bars represent the standard deviation.

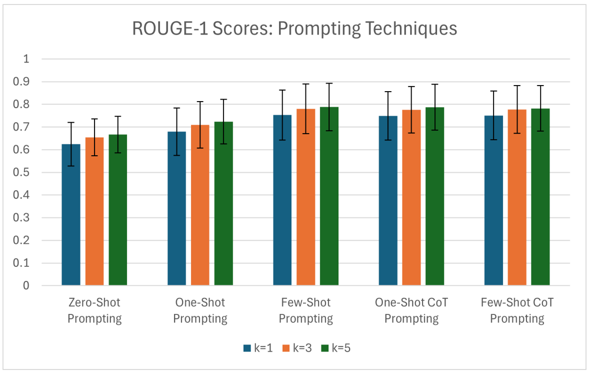


Figure 4.3: Bar chart with mean ROUGE-1 score of generated code across five different prompting techniques, with *k* values of 1, 3, and 5. Error bars represent the standard deviation.

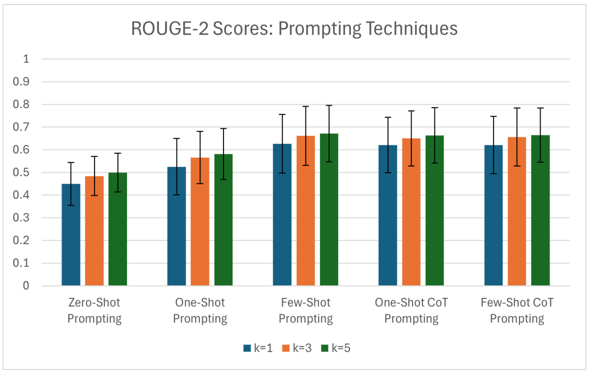


Figure 4.4: Bar chart with mean ROUGE-2 score of generated code across five different prompting techniques, with *k* values of 1, 3, and 5. Error bars represent the standard deviation.

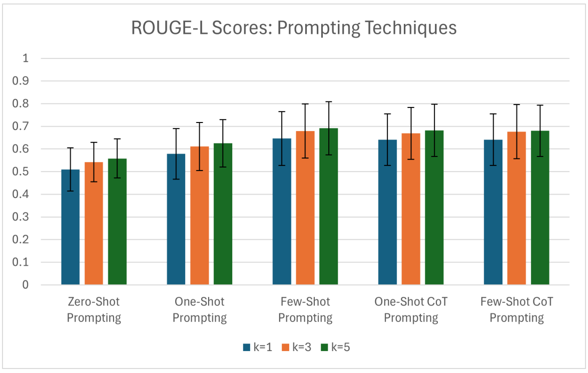


Figure 4.5: Bar chart with mean ROUGE-L score of generated code across five different prompting techniques, with *k* values of 1, 3, and 5. Error bars represent the standard deviation.

4.1.2. Execution-Based Metrics

Buildability

The build@k score of the generated code measures the proportion of generated garages that can be successfully built within the project. This score verifies the integrability of the generated code, ensuring its validity within the project's broader context and the successful resolution of all dependencies. Table 4.3 presents the build@k scores for experiment 1, comparing the five different prompting techniques across *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 4.3: build@k score of generated code across five different prompting techniques with *k* values of 1, 3, and 5.

Figure 4.6 presents the build@k scores from experiment 1 in a bar chart, providing a clear overview and allowing for comparisons between the different prompting techniques.

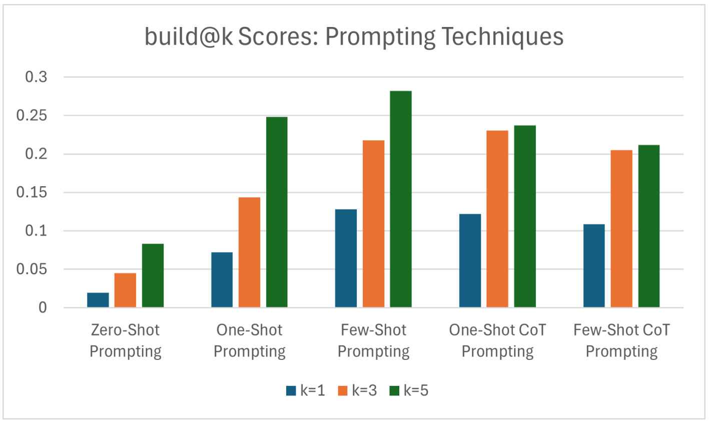


Figure 4.6: Bar chart with build@k score of generated code across five different prompting techniques, with k values of 1, 3, and 5.

The build@k scores presented in table 4.3 and figure 4.6 show considerable performance differences among the various prompting techniques. Out of the five prompting techniques, zero-shot prompting results in the lowest build@k score for all k values. The remaining techniques exhibit relatively comparable performance in terms of build@k scores. Few-shot prompting achieves the highest build score at $k = 1$ and $k = 5$. At $k = 1$, one-shot CoT prompting closely follows, with a very small difference. However, at $k = 5$, the gap between the top two techniques is quite large, with few-shot prompting generating 44 buildable garages and the second-best technique, one-shot prompting, producing 38. One-shot CoT prompting also demonstrates strong performance across all k values, outperforming all techniques at $k = 3$ by generating 36 garages that build successfully within the project.

Similar to the observation made for the match-based metric, the build@k score of the generated code across all prompting methods improves as k increases. This improvement is likely attributed to the fact that higher k values provide more opportunities for finding a buildable solution. The score difference between $k = 3$ and $k = 5$ is quite large for zero-shot, one-shot, and few-shot prompting while, the CoT prompting techniques exhibit minimal score differences between these k values.

Unit Tests

The second aspect of the execution-based metrics that we considered was the pass@k score of the generated code. To evaluate the generated code using unit tests, it must first successfully pass through the project's build pipeline. However, only a small proportion of the generated garages met this criterion, limiting the number of garages that could be tested.

Unfortunately, none of the eligible generated garages passed the unit tests, resulting in a pass@k score of 0.0 for all prompting techniques and k values, as shown in table 4.4. This outcome may be attributed to either the absence of unit tests for the generated garages or the failure of the generated garages to pass the existing unit tests.

	pass@k		
	k=1	*k=3*	*k=5*
Zero-Shot Prompting	0.000	0.000	0.000
One-Shot Prompting	0.000	0.000	0.000
Few-Shot Prompting	0.000	0.000	0.000
One-Shot CoT Prompting	0.000	0.000	0.000
Few-Shot CoT Prompting	0.000	0.000	0.000

Table 4.4: pass@k score of generated code across five different prompting techniques, with k values of 1, 3, and 5.

Table 4.5, presents the proportion of eligible garages that had missing unit tests versus those that had unit tests but failed to pass them. The analysis reveals that the majority of eligible garages did not have unit tests to verify their functional correctness. Specifically, the proportion of eligible garages with associated unit tests ranged from 0% to 21%, indicating a lack of test coverage.

		Zero-Shot	One-Shot	Few-Shot	One-Shot CoT	Few-Shot CoT
	# of successful garages built	3	11	20	19	17
***k* = 1**	**% of garages with no tests**	100%	100%	80%	79%	94%
	% of garages with tests	0%	0%	20%	21%	6%
	# of successful garages built	7	22	34	36	32
***k* = 3**	**% of garages with no tests**	86%	91%	88%	89%	84%
	% of garages with tests	14%	9%	12%	11%	16%
	# of successful garages built	13	38	44	37	33
***k* = 5**	**% of garages with no tests**	85%	84%	82%	81%	88%
	% of garages with tests	15%	16%	18%	19%	12%

Table 4.5: Proportion of built garages with and without unit tests across five different prompting techniques, with k values of 1, 3, and 5.

Code Quality

The final aspect of the execution-based metrics that we consider is the quality of the generated code. Tables 4.6, 4.7, and 4.8, presented below, outline six key metrics that we use to evaluate code quality of the generated code across the different prompting techniques and values of k. These metrics include the total number of violations across all built garages, the number of violations per built garage, the net change in violations (delta), the number of critical violations, number of critical violations per build and the number of non-critical violations.

	***k*=1**				
	Zero-Shot	**One-Shot**	**Few-Shot**	**One-Shot CoT**	**Few-Shot CoT**
Number of successful builds	3	11	20	19	17
Total Violations	6	14	21	19	17
Violations per build	2.00	1.27	1.05	**1.00**	**1.00**
Net (delta) Violations	2	-1	-9	-11	-9
Total Critical Violations	0	0	0	0	0
Critical violations per build	0	0	0	0	0
Total Non-Critical Violations	6	14	21	19	17

Table 4.6: Code quality metrics of generated code across five different prompting techniques at $k = 1$.

	***k*=3**				
	Zero-Shot	**One-Shot**	**Few-Shot**	**One-Shot CoT**	**Few-Shot CoT**
Number of successful builds	7	22	34	36	32
Total Violations	16	29	39	38	36
Violations per build	2.29	1.32	1.15	**1.06**	1.13
Net (delta) Violations	4	-6	-9	-16	-10
Total Critical Violations	1	1	1	1	2
Critical violations per build	0.143	0.045	0.029	**0.028**	0.063
Total Non-Critical Violations	15	28	38	37	34

Table 4.7: Code quality metrics of generated code across five different prompting techniques at $k = 3$.

	k=5				
	Zero-Shot	**One-Shot**	**Few-Shot**	**One-Shot CoT**	**Few-Shot CoT**
Number of successful builds	13	38	44	37	33
Total Violations	26	42	50	41	37
Violations per build	2.00	**1.11**	1.14	**1.11**	1.12
Net (delta) Violations	4	-8	-11	-12	-12
Total Critical Violations	0	0	2	2	2
Critical violations per build	0.000	0.000	0.045	0.054	0.061
Total Non-Critical Violations	26	42	48	39	35

Table 4.8: Code quality metrics of generated code across five different prompting techniques at $k = 5$.

The results from the tables above indicate that zero-shot prompting consistently generates low-quality code. For all three values of k, the average number of violations per build was 2 or more, whereas the other prompting techniques had a lower violation-per-build ratio, with a maximum of 1.32. Furthermore, the poor performance of zero-shot prompting is highlighted by the net number of violations, as it consistently introduced more code quality violations compared to the reference implementation. In contrast, the other prompting techniques consistently reduced the net number of violations. Overall, one-shot CoT prompting produced the highest-quality code across all k values, although the difference between few-shot and few-shot CoT prompting was also very small.

4.2. Experiment 2: Generic vs Code-Specific LLMs

Experiment 2 investigated the performance difference in code generation between generic LLMs and code-specific LLMs, addressing research question 2. This section reports the match-based and execution-based outcomes of experiment 2, with a detailed analysis and interpretation of these results provided in section 5.

4.2.1. Match-Based Metrics

Tables 4.9, 4.10 and 4.11 present the mean BLEU, CodeBLEU and ROUGE scores for the generated code, comparing the performance of generic and code-specific LLMs. To provide a comprehensive overview, three distinct model families were employed: Qwen2.5, Gemma, and DeepSeek-V2. Consistent with experiment 1, the mean BLEU, CodeBLEU and ROUGE scores were calculated using formula 4.1 to derive the values presented in the tables.

	BLEU					
	k=1		*k*=3		*k*=5	
Qwen2.5-7B-Instruct	**0.542**	-0.4%	0.590	+1.2%	0.597	+3.9%
Qwen2.5-Coder-7B-Instruct	0.540		**0.597**		**0.620**	
gemma-7b-it	0.198	+75.3%	0.261	+73.2%	0.273	+76.9%
codegemma-7b-it	**0.347**		**0.452**		**0.483**	
DeepSeek-V2-Lite-Chat	0.238	+109.7%	0.320	+70.6%	0.347	+62.8%
DeepSeek-Coder-V2-Lite-Instruct	**0.499**		**0.546**		**0.565**	

Table 4.9: Mean BLEU scores of generated code comparing generic and code-specific LLMs with *k* values of 1, 3, and 5.

	CodeBLEU					
	k=1		*k*=3		*k*=5	
Qwen2.5-7B-Instruct	**0.418**	-2.4%	0.455	+0.9%	**0.568**	-16.0%
Qwen2.5-Coder-7B-Instruct	0.408		**0.459**		0.477	
gemma-7b-it	**0.230**	-11.7%	0.263	+18.6%	0.279	+28.3%
codegemma-7b-it	0.203		**0.312**		**0.358**	
DeepSeek-V2-Lite-Chat	0.191	+105.8%	0.255	+68.2%	0.283	+55.8%
DeepSeek-Coder-V2-Lite-Instruct	**0.393**		**0.429**		**0.441**	

Table 4.10: Mean CodeBLEU scores of generated code comparing generic and code-specific LLMs with *k* values of 1, 3, and 5.

	ROUGE-1			ROUGE-2			ROUGE-L		
	k=1	*k*=3	*k*=5	*k*=1	*k*=3	*k*=5	*k*=1	*k*=3	*k*=5
Qwen2.5-7B-Instruct	0.704	0.735	0.742	0.561	0.599	0.607	0.593	0.626	0.636
Qwen2.5-Coder-7B-Instruct	**0.710**	**0.746**	**0.757**	**0.574**	**0.615**	**0.630**	**0.612**	**0.647**	**0.659**
gemma-7b-it	0.400	0.469	0.482	0.271	0.332	0.345	0.340	0.395	0.406
codegemma-7b-it	**0.558**	**0.633**	**0.664**	**0.394**	**0.472**	**0.508**	**0.457**	**0.523**	**0.553**
DeepSeek-V2-Lite-Chat	0.412	0.509	0.545	0.262	0.345	0.382	0.327	0.411	0.446
DeepSeek-Coder-V2-Lite-Instruct	**0.681**	**0.714**	**0.727**	**0.526**	**0.568**	**0.585**	**0.566**	**0.601**	**0.614**

Table 4.11: Mean ROUGE-1, ROUGE-2, ROUGE-L scores of generated code comparing generic and code-specific LLMs with *k* values of 1, 3, and 5.

The tables above provide interesting insights into the similarity of generated code from code-specific and generic LLMs. The results show that code generated by code-specific LLMs is generally more similar to the reference code than the code produced by generic LLMs. However, the difference in similarity varies across model families. The Qwen2.5 models exhibit a negligible difference in similarity between the code generated by the generic and code-specific variants. In contract, the Gemma and DeepSeek-V2 models display a substantially larger difference in similarity, suggesting that the benefits of code-specific models are greater in these model families.

The similarity scores for the code generated by the Qwen2.5-7B-Instruct and Qwen2.5-Coder-7B-Instruct models are remarkably close, suggesting that both models may be equally effective for generating domain-specific code. In terms of BLEU scores, the code-specific model generally performed the best, except for at $k = 1$, where the generic model's code was slightly more similar to the reference code. The largest difference in BLEU score was observed at $k = 5$, where the code-specific model had a higher similarity score by 0.023. In terms of CodeBLEU scores, the generic model performed better than the code-specific model at $k = 1$ and $k = 5$ and marginally worse at $k = 3$. The largest difference in score was again observed at $k = 5$. For ROUGE scores, the code-specific model consistently outperformed the generic model, although the difference was once again very small.

In contrast to the Qwen2.5 models, the Gemma and DeepSeek-V2 models showed a larger difference in similarity scores between code generated by code-specific and generic model variants. For the Gemma family, the code specific variant outperformed the generic variant in all metrics across all values of k except for the CodeBLEU score at $k = 1$. The code-specific variant of the DeepSeek-V2 family consistently generated code with a higher similarity score than their generic counterparts, across all values of k.

The DeepSeek-V2 models exhibited a larger average difference in similarity compared to the Gemma models. Specifically, the average increase in BLEU similarity for the DeepSeek-V2 models was 81.03%, whereas for the Gemma models it was 75.13%. The average increase in CodeBLEU score was slightly smaller but still more pronounced for the DeepSeek-V2 models, at 76.6%, compared to 19.5% for the Gemma models. These findings suggest that the code-specific models from these families have a substantial advantage over their generic counterparts in generating domain-specific code, with the DeepSeek-V2 models appearing to benefit more from code-specific fine-tuning.

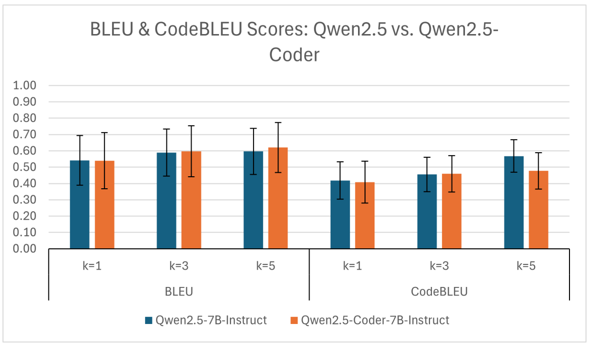


Figure 4.7: Bar chart with mean BLEU and CodeBLEU scores of generated code for Qwen2.5-7B-Instruct and Qwen2.5-Coder-7B-Instruct, with k values of 1, 3, and 5. Error bars represent the standard deviation.

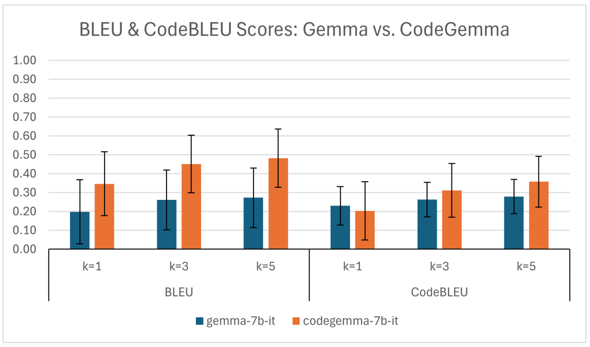


Figure 4.8: Bar chart with mean BLEU and CodeBLEU scores of generated code for gemma-7b-it and codegemma-7b-it, with k values of 1, 3, and 5. Error bars represent the standard deviation.

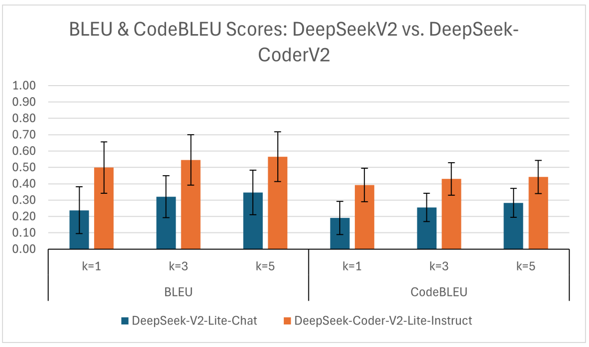


Figure 4.9: Bar chart with mean BLEU and CodeBLEU scores of generated code for DeepSeek-V2-Lite-Chat and DeepSeek-Coder-V2-Lite-Instruct, with k values of 1, 3, and 5. Error bars represent the standard deviation.

Figures 4.7, 4.8, and 4.9 present bar charts of the BLEU and CodeBLEU scores for the generated code from the three model families used in this experiment. These visualizations reinforce the previous findings, showing that the difference in BLEU and CodeBLEU scores between the generic and code-specific models is almost negligible for the Qwen2.5 family. In contrast, the difference in scores between the generic and code-specific models for the Gemma and DeepSeek-V2 family is larger, with the code-specific models almost always outperforming their generic counterparts.

Figures 4.10, 4.11, and 4.12 display bar charts of the ROUGE-1, ROUGE-2, and ROUGE-L scores from experiment 2. These visualizations confirm the previous observations, providing further evidence of the trends in code generation performance of domain and code-specific models. Additionally, the figures reveal that the similarity scores of the generated code tend to improve as the number of generations (k) increases, a pattern that was also observed in experiment 1.

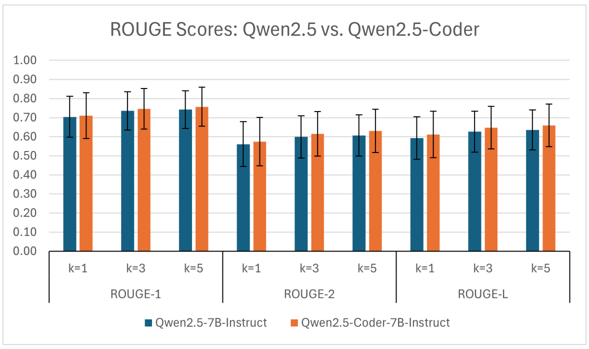


Figure 4.10: Bar chart with mean ROUGE-1, ROUGE-2, and ROUGE-L scores of generated code for Qwen2.5-7B-Instruct and Qwen2.5-Coder-7B-Instruct, with k values of 1, 3, and 5. Error bars represent the standard deviation.

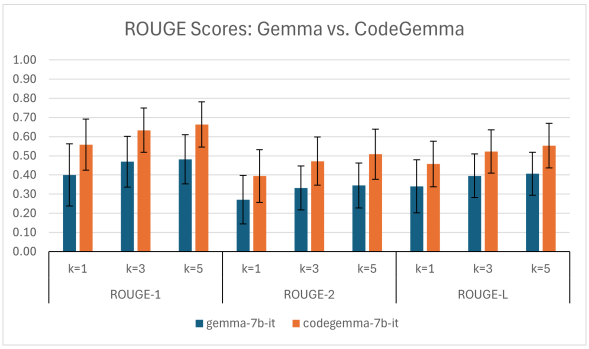


Figure 4.11: Bar chart with mean ROUGE-1, ROUGE-2, and ROUGE-L scores of generated code for gemma-7b-it and codegemma-7b-it, with k values of 1, 3, and 5. Error bars represent the standard deviation.

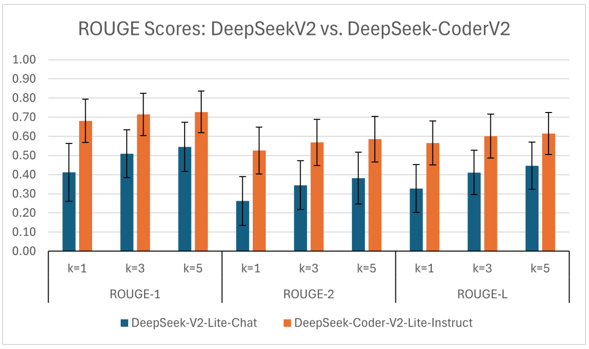


Figure 4.12: Bar chart with mean ROUGE-1, ROUGE-2, and ROUGE-L scores of generated code for DeepSeek-V2-Lite-Chat and DeepSeek-Coder-V2-Lite-Instruct, with k values of 1, 3, and 5. Error bars represent the standard deviation.

4.2.2. Execution-Based Metrics

Buildability

For the generated code to be usable in a real-world setting, it must be compatible with the rest of the project and allow for successful compilation. The buildability of the code is assessed using the build@k metric. The build@k scores for the generated code from the various models are presented in Table 4.12 below, spanning k values of 1, 3, and 5.

	build@k					
	k=1		*k*=3		*k*=5	
Qwen2.5-7B-Instruct	0.019	+68.4%	0.058	+10.3%	0.038	+86.8%
Qwen2.5-Coder-7B-Instruct	**0.032**		**0.064**		**0.071**	
gemma-7b-it	0.006	+4916.7%	0.013	+3546.2%	0.019	+2800%
codegemma-7b-it	**0.301**		**0.474**		**0.551**	
DeepSeek-V2-Lite-Chat	**0.071**	-73.2%	**0.147**	-69.4%	**0.205**	-71.7%
DeepSeek-Coder-V2-Lite-Instruct	0.019		0.045		0.058	

Table 4.12: build@k score of generated code, comparing generic and code-specific LLMs with *k* values of 1, 3, and 5.

Similar to the match-based metrics, table 4.12 shows that the build@k scores of the two Qwen2.5 models are relatively close. Although the code-specific Qwen2.5 model slightly outperforms the generic model, for all values of k, the margin of difference is minute. The code-specific model achieved its highest build@k score at $k = 5$ by generating 11 buildable garages, whereas the generic model reached its highest score at $k = 3$ by generating 9 buildable garages.

The difference in build@k scores between the generic and code-specific Gemma models is large, with the code-specific model clearly outperforming the generic model in terms of generating buildable code. The build@k score reveals a substantial performance gap between the two models, which is consistent with the similarity results that already suggested the code-specific model's superiority. At $k = 5$, the point at which both models produced the most buildable code, the generic model generated only 3 buildable garages, whereas the code-specific model produced 86. This significant difference is consistent across all other values of k, with the code-specific model consistently generating a substantially larger number of buildable garages than its generic counterpart.

For the DeepSeek-V2 models, the code-specific model had a lower build@k score than the generic model, which was unexpected. Although the code-specific DeepSeek model produced more similar code, its build@k score exhibited an opposing trend. This discrepancy underscores that the similarity score is insufficient as the only metric for evaluating generated code performance. Across all k values, the code-specific model generated fewer buildable code instances than the generic model, potentially indicating that the generic DeepSeek-V2 model is more suitable for domain-specific code generation.

Figure 4.13 presents the build@k scores from experiment2 in a bar chart providing a visual overview of the build performance of the generated code.

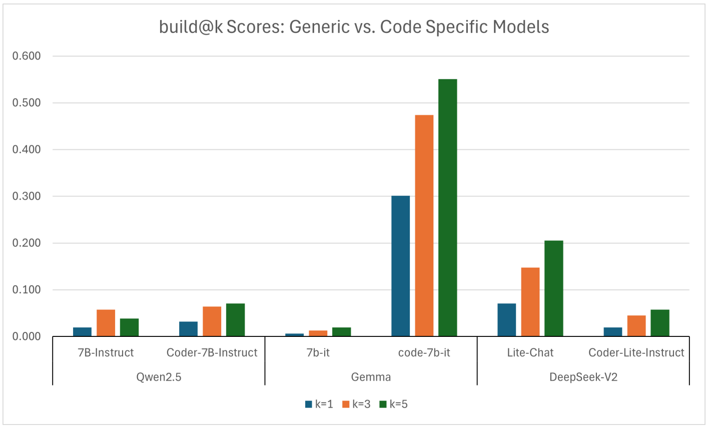


Figure 4.13: Bar chart with build@k score of generated code comparing generic and code-specific LLMs for the Qwen2.5, Gemma and DeepSeek-V2 families, with *k* values of 1, 3, and 5.

Unit Tests

We evaluate the functional correctness of the generated code by utilizing the existing unit tests within the codebase. According to the results presented in table 4.13, none of the generated garages successfully passed the unit tests.

	pass@k		
	k=1	*k=3*	*k=5*
Qwen2.5-7B-Instruct	0.000	0.000	0.000
Qwen2.5-Coder-7B-Instruct	0.000	0.000	0.000
gemma-7b-it	0.000	0.000	0.000
codegemma-7b-it	0.000	0.000	0.000
DeepSeek-V2-Lite-Chat	0.000	0.000	0.000
DeepSeek-Coder-V2-Lite-Instruct	0.000	0.000	0.000

Table 4.13: pass@k score of generated code, comparing generic and code-specific LLMs with *k* values of 1, 3, and 5.

There could be two main reasons that the pass@k score for all models is 0. The first possibility is that the generated garage lacks an existing unit test in the repository. The second possibility is that the generated garage simply failed to pass the associated unit tests. Table 4.14 provides a detailed breakdown of the proportion of garages that have unit tests and those that do not, offering insight into the distribution of test coverage among the generated buildable garages.

		Qwen2.5		*Gemma*		*DeepSeek-V2*	
		Generic	**Code**	**Generic**	**Code**	**Generic**	**Code**
	# of successful garages built	3	5	1	47	11	3
k* = *1	**% of garages with no tests**	100%	60%	0%	68%	73%	67%
	% of garages with tests	0%	40%	100%	32%	27%	33%
	# of successful garages built	9	10	2	74	23	7
k* = *3	**% of garages with no tests**	67%	60%	50%	77%	70%	71%
	% of garages with tests	33%	40%	50%	23%	30%	29%
	# of successful garages built	6	11	3	86	32	9
k* = *5	**% of garages with no tests**	67%	73%	33%	74%	75%	89%
	% of garages with tests	33%	27%	67%	26%	25%	11%

Table 4.14: Proportions of built garages with and without unit tests, comparing generic and code-specific LLMs with *k* values of 1, 3, and 5.

The breakdown from table 4.14 highlights the lack of unit test coverage in the generated garages for all model families. Since many of the generated garages lacked unit tests, it is not possible to make a definitive statements regarding the functional correctness of the generated code. For the Qwen2.5 family of models, only 28% of the generated buildable garages had unit tests presents. Similarly for the buildable garages generated by the DeepSeek-V2 models only 26% had existing unit tests. In contrast, the Gemma models achieved the highest proportion of buildable garages with unit tests, at approximately 50%. Despite this relatively higher coverage, none of the generated code successfully passed the unit tests, highlighting a gap in the functional correctness of the generated garages.

Code Quality

The final quantitative aspect of the generated code that we consider is the code quality. Figures 4.15, 4.16 and 4.17 provide a breakdown of the code quality of the generated code from the three model families across six key metrics: the total number of violations across all built garages, the number of violations per built garage, the net change in violations (delta), the number of critical violations, number of critical violations per build and the number of non-critical violations.

Generic = Qwen2.5-7B-Instruct	***k=1***		***k=3***		***k=5***	
Code-Specific = Qwen2.5-Coder-7B-Instruct	**Generic**	**Code**	**Generic**	**Code**	**Generic**	**Code**
Number of successful builds	3	5	9	10	6	11
Total Violations	5	5	15	10	10	12
Violations per Build	1.67	**1.00**	1.67	**1.00**	1.67	**1.09**
Net (delta) Violations	2	-2	3	-5	2	-4
Total Critical Violations	0	0	0	0	0	0
Critical Violations per Build	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**
Total Non-Critical Violations	5	5	15	10	10	12

Table 4.15: Code quality metrics of generated code, comparing Qwen2.5-7B-Instruct and Qwen2.5-Coder-7B-Instruct with k values of 1, 3, and 5.

The results presented in table 4.15 indicate that the code-specific Qwen2.5 model generates code of higher quality compared to the generic model. The generic model consistently exhibits a violations per build ratio of 1.67, whereas the code-specific model achieves a ratio close to 1.00. Both models perform equally well in terms of critical violations, as neither model incurred any critical violations.

Generic = gemma-7b-it	***k=1***		***k=3***		***k=5***	
Code-Specific = codegemma-7b-it	**Generic**	**Code**	**Generic**	**Code**	**Generic**	**Code**
Number of successful builds	1	47	2	74	3	86
Total Violations	2	169	4	254	6	275
Violations per Build	**2.00**	3.60	**2.00**	3.43	**2.00**	3.20
Net (delta) Violations	1	61	0	102	3	105
Total Critical Violations	0	34	0	48	0	48
Critical Violations per Build	**0.000**	0.723	**0.000**	0.649	**0.000**	0.558
Total Non-Critical Violations	2	135	4	206	6	227

Table 4.16: Code quality metrics of generated code, comparing gemma-7b-it and codegemma-7b-it with k values of 1, 3, and 5.

The code quality metrics for the two Gemma models are presented in Table 4.16. The results indicate that the code-specific model struggles to generate high-quality code compared to the generic model. The generic model maintains a consistent violation per build ratio of 2.00, whereas the code-specific model's ratio exceeds 3.20 for all values of k. Furthermore, the generic model achieves a critical violation per build ratio of 0.00, while the code-specific model's ratio remains above 0.55. However, it's essential to consider that the generic model's sample size is relatively small, and its performance may deteriorate if it generates a larger number of garages.

Generic = DeepSeek-V2-Lite-Chat	***k=1***		***k=3***		***k=5***	
Code-Specific = DeepSeek-Coder-V2-Lite-Instruct	**Generic**	**Code**	**Generic**	**Code**	**Generic**	**Code**
Number of successful builds	11	3	23	7	32	9
Total Violations	36	6	74	10	97	14
Violations per Build	3.27	**2.00**	3.22	**1.43**	3.03	**1.56**
Net (delta) Violations	9	2	26	1	35	5
Total Critical Violations	6	0	13	0	15	0
Critical Violations per Build	0.55	**0.00**	0.57	**0.00**	0.47	**0.00**
Total Non-Critical Violations	30	6	61	10	82	14

Table 4.17: Code quality metrics of generated code, comparing DeepSeek-V2-Lite-Chat and DeepSeek-Coder-V2-Lite-Instruct with k values of 1, 3, and 5.

Finally, table 4.17 summarizes the code quality of the code generated by the DeepSeek-V2 models. The results show that the code-specific model generates higher-quality code than the generic model. The generic models consistently exhibit a violations per build ratio exceeding 3.00, whereas the code-specific model maintains a ratio below 2.00. Furthermore, the code-specific model has no critical violations per build, in contrast to the generic model, which has a ratio of approximately 0.5 per build.

4.3. Experiment 3: Model Sizes

Experiment 3 investigated the impact of differing models sizes on the quality of generated code. This section presents the results of the experiments conducted for experiment 3. Both match-based and execution based outcomes are presented in this section.

4.3.1. Match-Based Metrics

Tables 4.18, 4.19 and 4.20 present the mean BLEU, CodeBLEU and ROUGE scores of the generated code for different model sizes of Qwen2.5-Coder-Instruct and k values of 1, 3, and 5. Similar to the previous experiments, formula 4.1 was used to compute the mean values.

	BLEU					
	k=1		*k*=3		*k*=5	
Qwen2.5-Coder-0.5B-instruct	0.183		0.263		0.292	
Qwen2.5-Coder-1.5B-instruct	0.239	+30.6%	0.354	+34.6%	0.406	+39.0%
Qwen2.5-Coder-3B-instruct	0.390	+63.2%	0.488	+37.9%	0.532	+31.0%
Qwen2.5-Coder-7B-instruct	0.540	+38.5%	0.597	+22.3%	0.620	+16.5%
Qwen2.5-Coder-14B-instruct	0.605	+12.0%	0.638	+6.9%	0.658	+6.1%
Qwen2.5-Coder-32B-instruct	**0.615**	+1.7%	**0.652**	+2.2%	**0.670**	+1.8%

Table 4.18: Mean BLEU scores of generated code across different model sizes of Qwen2.5-Coder-Instruct, for k values of 1, 3, and 5.

	CodeBLEU					
	k=1		*k*=3		*k*=5	
Qwen2.5-Coder-0.5B-instruct	0.163		0.247		0.278	
Qwen2.5-Coder-1.5B-instruct	0.194	+19.0%	0.283	+14.6%	0.311	+11.9%
Qwen2.5-Coder-3B-instruct	0.303	+56.2%	0.378	+33.6%	0.406	+30.5%
Qwen2.5-Coder-7B-instruct	0.408	+34.7%	0.459	+21.4	0.477	+17.5%
Qwen2.5-Coder-14B-instruct	0.455	+11.5%	0.492	+7.2%	0.502	+5.2%
Qwen2.5-Coder-32B-instruct	**0.456**	+0.2%	**0.502**	+2.0%	**0.512**	+2.0%

Table 4.19: Mean CodeBLEU scores of generated code across different model sizes of Qwen2.5-Coder-Instruct, for k values of 1, 3, and 5.

	ROUGE-1			**ROUGE-2**			**ROUGE-L**		
	k=1	*k*=3	*k*=5	*k*=1	*k*=3	*k*=5	*k*=1	*k*=3	*k*=5
Qwen2.5-Coder-0.5B-instruct	0.329	0.403	0.426	0.170	0.234	0.254	0.267	0.325	0.343
Qwen2.5-Coder-1.5B-instruct	0.443	0.559	0.600	0.309	0.407	0.449	0.375	0.492	0.503
Qwen2.5-Coder-3B-instruct	0.584	0.657	0.688	0.437	0.513	0.545	0.476	0.534	0.561
Qwen2.5-Coder-7B-instruct	0.710	0.746	0.757	0.574	0.615	0.630	0.612	0.647	0.659
Qwen2.5-Coder-14B-instruct	0.747	0.771	0.782	0.619	**0.650**	0.661	**0.646**	**0.675**	**0.684**
Qwen2.5-Coder-32B-instruct	**0.749**	**0.776**	**0.787**	**0.620**	**0.650**	**0.663**	0.641	0.669	0.682

Table 4.20: Mean ROUGE-1, ROUGE-2, ROUGE-L scores of generated code across different model sizes of Qwen2.5-Coder-Instruct, for k values of 1, 3, and 5.

The results presented in the three tables above, demonstrate that the model size has a large impact on performance, with larger models generally achieving higher code similarity scores across all metrics. This suggests that increasing the model size can lead to improved performance, as the model is able to capture more complex patterns and relationships in the data. Specifically, the 32B model emerges as a top performer, consistently obtaining the highest scores for BLEU, CodeBLEU and ROUGE across all evaluations when tasked with generating domain specific code.

In terms of the effect of model size, the results show an upward trend in code similarity scores as the number of model parameters increases. The smallest model size, 0.5 billion parameters, exhibits the lowest similarity, while the larger models, particularly 14 and 32 billion parameters, demonstrate high similarity with the reference solution. The improvement in code similarity scores is largest as the model size increases from 0.5B to 3B, with a substantial jump in scores across all measurements. However, the difference in code similarity scores between the 14B and 32B model is very minimal, suggesting that the rate of improvement may be reducing as the model size increases after a certain point.

Overall, the Qwen2.5-Coder-32B-Instruct model produced code with the highest similarity to the reference solution. The 14B model also generated code with very high similarity, narrowly trailing the 32B model in most cases, and even outperforming it in terms of ROUGE-L metric.

Similar to the observations made in the previous experiments, the value of k, also has an impact on the similarity between the generated code and the reference code, although to a lesser extent. The difference in code similarity between $k = 1$ and $k = 3$ is quite large, however the difference between $k = 3$ and $k = 5$ is less pronounced.

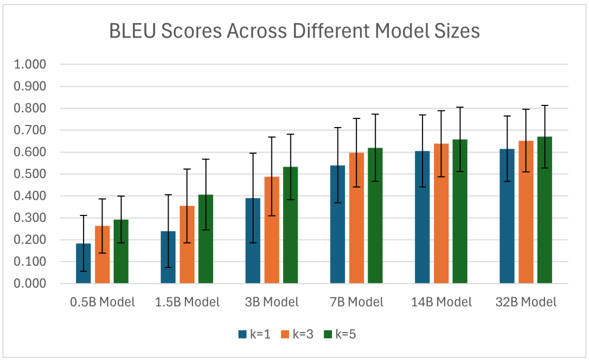


Figure 4.14: Bar chart with mean BLEU scores of generated code across different model sizes of Qwen2.5-Coder-Instruct, with k values of 1, 3, and 5. Error bars represent the standard deviation.

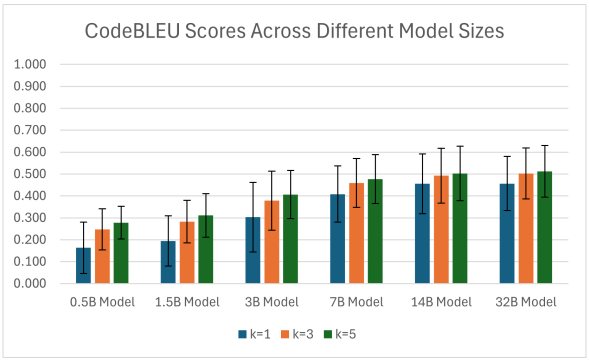


Figure 4.15: Bar chart with mean CodeBLEU scores of generated code across different model sizes of Qwen2.5-Coder-Instruct, with k values of 1, 3, and 5. Error bars represent the standard deviation.

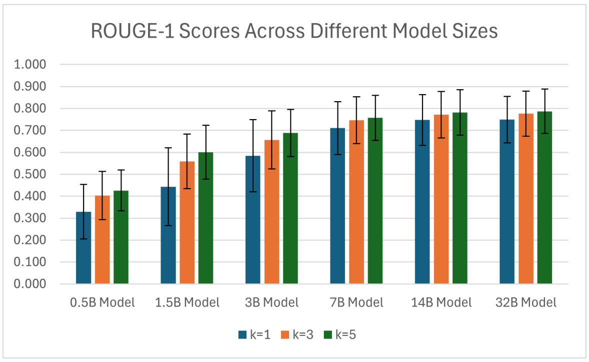


Figure 4.16: Bar chart with mean ROUGE-1 scores of generated code across different model sizes of Qwen2.5-Coder-Instruct, with k values of 1, 3, and 5. Error bars represent the standard deviation.

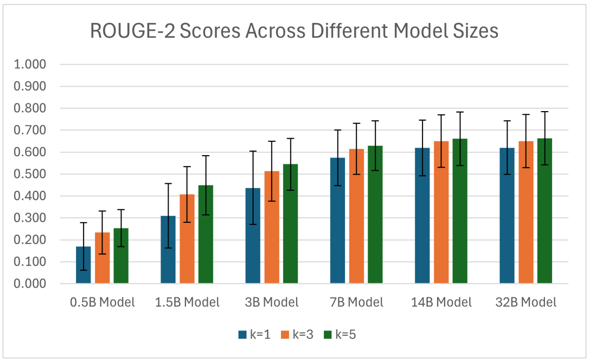


Figure 4.17: Bar chart with mean ROUGE-2 scores of generated code across different model sizes of Qwen2.5-Coder-Instruct, with k values of 1, 3, and 5. Error bars represent the standard deviation.

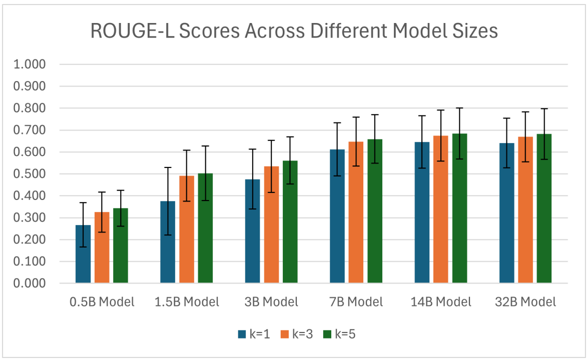


Figure 4.18: Bar chart with mean ROUGE-L scores of generated code across different model sizes of Qwen2.5-Coder-Instruct, with k values of 1, 3, and 5. Error bars represent the standard deviation.

Figures 4.14, 4.15, 4.16, 4.17, and 4.18 offer a visual representation of the similarity scores for different model sizes of Qwen2.5-Coder-Instruct. The charts illustrate the performance differences between the models, with the smaller 0.5B model consistently scoring lower than the larger models. In contrast, the 14B and 32B models demonstrate higher similarity scores, indicating their superior performance. The visualizations also suggest that the relationship between model size and similarity scores is not entirely linear, as the gains in similarity scores become less pronounced as the model size increases. This is evident in the relatively smaller differences in scores between the 14B and 32B models. Overall, the figures provide a clear visual representation of the impact of model size on similarity of generated code.

The error bars in the bar charts denote the standard deviation of the similarity metrics across all 156 generated garages. The standard deviation is relatively large for smaller models, suggesting a high degree of variability in their performance. This is particularly evident for models with lower mean scores, where the coefficient of variation is quite large. As the model sizes increase, the error bars also become smaller indicating a smaller spread of the data with respect to the mean.

4.3.2. Execution-Based Metrics

Buildability

Table 4.21 shows that the build@k score across the six different model sizes differs substantially. Unlike the continuous upward trend that was observed in the match-based metrics, the build@k scores do not have a continuous upward trend and differ quite a lot between the different model sizes.

	build@k					
	k=1		*k=3*		*k=5*	
Qwen2.5-Coder-0.5B-instruct	0.00000		0.00641		0.00641	
Qwen2.5-Coder-1.5B-instruct	0.10897		**0.23077**	+3500%	**0.34615**	+5300%
Qwen2.5-Coder-3B-instruct	0.08333	-23.5%	0.14102	-38.9%	0.18589	-46.3%
Qwen2.5-Coder-7B-instruct	0.03205	-61.5%	0.06410	-51.5%	0.07051	-62.1%
Qwen2.5-Coder-14B-instruct	**0.12179**	+280%	0.21154	+230%	0.27561	+290%
Qwen2.5-Coder-32B-instruct	**0.12179**	+0%	**0.23077**	+9.1%	0.23718	-13.9%

Table 4.21: build@k score of generated code across different model sizes of Qwen2.5-Coder-Instruct, with k values of 1, 3, and 5.

Figure 4.19 presents the build@k score for experiment 3 in a bar chart which compares the buildability of the generated code across the six different model sizes with k values of 1, 3, and 5.

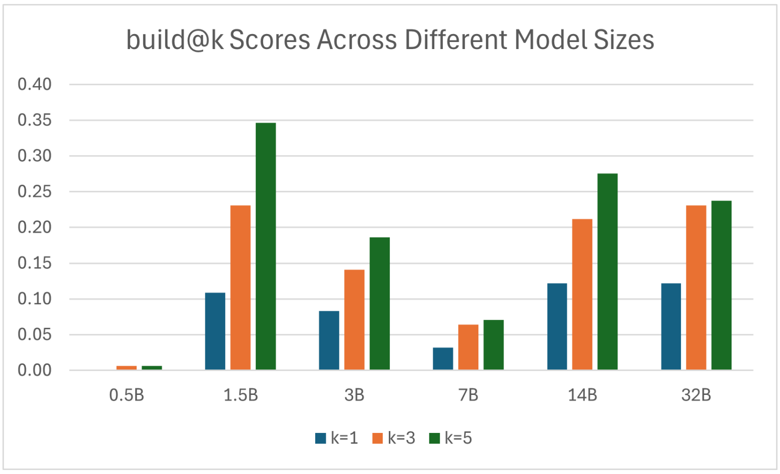


Figure 4.19: Bar chart with build@k score of generated code across different model sizes of Qwen2.5-Coder-Instruct, with k values of 1, 3, and 5.

The build@k scores presented in Table 4.21 and Figure 4.19 show that there is a large disparity in the buildability of generated code across different model sizes. The 0.5B model struggled the most, failing to generate a single buildable garage at $k = 1$ and only achieving one successful build at $k = 3$ and $k = 5$. The second smallest model, *Qwen2.5-Coder-1.5B-instruct*, demonstrated surprisingly strong performance, as it generated many buildable garages. For $k = 5$ it even managed to generate the most garages that had a successful build even outperforming larger models like the 32B and 14B models. At $k = 3$, the 1.5B model achieved identical results to the largest 32B model.

The 3B and 7B models showed poor performance in comparison to the 1.5B model, which was unexpected given their larger size. The 7B model, in particular, performed the worst after the 0.5B model in terms of generating buildable code. On the other hand, the 14B and 32B models demonstrated strong performance, achieving the highest build score at $k = 1$. Interestingly, the difference in performance between the 14B and 32B models was relatively small, suggesting that model size may not have an impact in this case.

As observed for the match-based metrics the build@k score improves as k increases for all model sizes, except for the 0.5B model. This improvement is likely due to the fact that a larger value for k allows the models with more opportunities to generate code that builds successfully. However, the 0.5B model appears to be an outlier, failing to capitalize on these opportunities.

Unit Tests

Amongst all garages that were successfully built, none of them managed to pass the unit as shown in table 4.22.

	pass@k		
	k=1	*k=3*	*k=5*
Qwen2.5-Coder-0.5B-instruct	0.0	0.0	0.0
Qwen2.5-Coder-1.5B-instruct	0.0	0.0	0.0
Qwen2.5-Coder-3B-instruct	0.0	0.0	0.0
Qwen2.5-Coder-7B-instruct	0.0	0.0	0.0
Qwen2.5-Coder-14B-instruct	0.0	0.0	0.0
Qwen2.5-Coder-32B-instruct	0.0	0.0	0.0

Table 4.22: pass@k score of generated code across different model sizes of Qwen2.5-Coder-Instruct, for k values of 1, 3, and 5.

The fact that no garages passed the unit tests could be attributed to either a lack of unit tests or a failure to meet the test requirements. Table 4.23 indicates the proportion of eligible garages that do not have an associated unit.

		Qwen2.5-Coder-Instruct					
		0.5B	**1.5B**	**3B**	**7B**	**14B**	**32B**
***k* = 1**	**# of successful garages built**	0	17	13	5	19	19
	% of garages with no tests	0%	82%	69%	60%	84%	79%
	% of garages with tests	0%	18%	31%	40%	16%	21%
***k* = 3**	**# of successful garages built**	1	36	22	10	33	36
	% of garages with no tests	100%	81%	50%	60%	79%	89%
	% of garages with tests	0%	19%	50%	40%	21%	11%
***k* = 5**	**# of successful garages built**	1	54	29	11	43	37
	% of garages with no tests	0%	81%	69%	73%	81%	81%
	% of garages with tests	100%	19%	31%	27%	19%	19%

Table 4.23: Proportion of built garages with and without unit tests, across different model sizes of Qwen2.5-Coder-Instruct, with k values of 1, 3, and 5.

Table 4.23 once again reveals that a significant proportion of the built garages do not have associated unit tests to verify the functional correctness of the generated code. On average, only 27% of the built garages had a corresponding unit test making it very hard to draw conclusions regarding the functional correctness of the generated code.

Code Quality

The final aspect we consider for experiment 3 is the code quality of the generated code. Tables 4.24, 4.25, and 4.26 present the six key metrics used to assess the code quality of the buildable garages generated in this experiment.

k=1	*Qwen2.5-Coder-Instruct*					
	0.5B	**1.5B**	**3B**	**7B**	**14B**	**32B**
Number of Successful Builds	0	17	13	5	19	19
Total Violations	0	52	30	5	20	19
Violations per Build	N/A	3.06	2.31	**1.00**	1.05	**1.00**
Net (delta) Violations	0	6	7	-2	-8	-11
Total Critical Violations	0	8	2	0	0	0
Critical Violations per Build	N/A	0.471	0.154	**0.000**	**0.000**	**0.000**
Total Non-Critical Violations	0	44	28	5	20	19

Table 4.24: Code quality metrics of generated code across different model sizes of Qwen2.5-Coder-Instruct, at $k = 1$

k=3	*Qwen2.5-Coder-Instruct*					
	0.5B	**1.5B**	**3B**	**7B**	**14B**	**32B**
Number of Successful Builds	1	36	22	10	33	36
Total Violations	2	101	81	10	36	38
Violations per Build	2	2.81	3.68	**1.00**	1.09	1.06
Net (delta) Violations	1	31	34	-5	-17	-16
Total Critical Violations	0	15	20	0	0	1
Critical Violations per Build	**0.000**	0.417	0.909	**0.000**	**0.000**	0.028
Total Non-Critical Violations	2	86	61	10	36	37

Table 4.25: Code quality metrics of generated code across different model sizes of Qwen2.5-Coder-Instruct, at $k = 3$

k=5	*Qwen2.5-Coder-Instruct*					
	0.5B	**1.5B**	**3B**	**7B**	**14B**	**32B**
Number of Successful Builds	1	54	29	11	43	37
Total Violations	4	182	76	12	48	41
Violations per Build	4	3.37	2.62	**1.09**	1.12	1.11
Net (delta) Violations	2	64	19	-4	-19	-12
Total Critical Violations	1	33	9	0	1	2
Critical Violations per Build	1.000	0.611	0.310	**0.000**	0.023	0.054
Total Non-Critical Violations	3	149	67	12	47	39

Table 4.26: Code quality metrics of generated code across different model sizes of Qwen2.5-Coder-Instruct, at $k = 5$

The code quality evaluation, as presented above, reveals that smaller models produce lower-quality code compared to their larger counterparts. Particularly, the 0.5B, 1.5B, and 3B models generate code with significant quality issues, as reflected in both the violations per build ratio and the critical violations per build ratio. These three models have a violations per build ratio exceeding 2 across all k values, accompanied by a relatively high critical violations per build ratio. In contrast, the larger models (7B, 14B, and 32B) demonstrate substantially fewer total and critical violations. The violations per build ratio for these larger models consistently is around 1.00, and they all display a very low critical violations per build ratio, indicating better code quality. The smaller model also consistently generates more code quality violations than the reference solution. as evidenced by the net (delta) violations metric, which remains consistently positive. The larger models, on the other hand, tend to always improve on the code quality in comparison to the reference solution, as indicated by the negative delta violations.

5 Discussion

This section presents an analysis and reflection of the experimental results, addressing the research questions posed earlier. We also examine potential threats to the validity of our experiments, providing a comprehensive evaluation of our findings.

5.1. RQ1: Prompting techniques and the impact thereof on the generated code

The results from experiment 1 (see 4.1), show that the prompting technique can have a substantial impact on the domain specific code generation capabilities of LLMs. Zero-shot prompting resulted in the poorest performance across all metrics. While one-shot prompting showed some improvement over zero-shot prompting, its performance still lagged behind the other techniques. In contrast, few-shot, one-shot CoT, and few-shot CoT prompting performed the best, with very little variation in between their scores.

The poor performance of the zero-shot prompting technique can be attributed to the absence of an example garage. With zero-shot prompting the model was only provided with the task description without any supporting examples, resulting in lower scores across the various metrics. With the other prompting techniques the model was explicitly provided with examples of one or more garages, allowing it to infer task patterns and do in-context learning, resulting in a higher performance of the generated code. This performance gap highlights the importance of providing examples especially when generating domain specific code in a specialized and unseen environment.

The experimental results also suggest a trend between the one-shot and few-shot prompting techniques. Across both match-based and execution-based metrics, the code generated by few-shot prompting consistently outperforms that of one-shot prompting. Although both techniques leverage in-context learning, the results indicate that the model benefits from being provided with more than one example. Few-shot prompting offers a more diverse set of patterns, increasing the likelihood that at least one example will closely align with the current problem. This enables the model to generalize more effectively, leading to improvements not only in similarity scores but also in the build@k score. The increase in scores highlights the importance of providing multiple examples in the prompt, particularly for domain-specific use cases where it is important for the model to generalize well.

Providing more than one example garage in the prompt did not improve the performance of the generated code when using CoT prompting. In terms of match-based metrics, few-shot CoT generally outperformed one-shot CoT, although the improvement in performance was smaller when compared to the regular one-shot and few-shot prompting techniques. Interestingly, the opposite trend was observed for build@k scores, where one-shot CoT prompting surpassed few-shot CoT prompting. This trend can be attributed to the possibility that providing the model with multiple explanations may lead to confusion. Alternatively, it can also be said that with multiple chains of thought, the model may start mimicking the examples too literally rather than generalizing. This suggests that the model's ability to effectively utilize CoT prompting may be sensitive to the number of examples provided, and that a single, well-crafted example may be more beneficial than multiple examples in certain cases.

The number of generated outputs, k, also has an impact on overall performance. Generating multiple outputs per task increases the likelihood of obtaining at least one high-quality solution, especially in probabilistic models like LLMs. This observation is consistent with findings from other LLM-based code generation studies [55], demonstrating its applicability in domain-specific settings as well. Interestingly, in terms of build@k scores, the improvement from $k = 1$ to $k = 3$ is more substantial than from $k = 3$ to $k = 5$. For the CoT prompting methods, the performance gain from increasing k beyond 3 is almost negligible. In industrial settings, such as those of ASML, it's not always necessary to generate the solution in a single attempt. If multiple generations can produce functional and high-quality code, it can still provide substantial business value.

Figures 5.1, 5.2, 5.3, 5.4, and 5.5 show the distribution of built garages by the number of context files, across the five prompting techniques, at $k = 5$. These histograms offer valuable insights into the relationship between generating buildable garages and garage complexity (in terms of context files). Interestingly, all the histograms are right-skewed, with the peak located on the left side of the graph. This skewness indicates that most of the generated garages that are buildable have relatively low complexity, with fewer than 10 context files.

Across all prompting techniques, the majority of buildable garages are generated when the number of context files ranges from 0 to 5. The second-highest number of buildable garages is generated when the number of context files falls between 5 and 10. In rare cases, the model successfully generates a buildable garage with more than 15 associated context files. In conclusion, the model is more successful at generating buildable garages for those with fewer context files. As the number of context files increases, the likelihood of the model generating buildable code decreases.

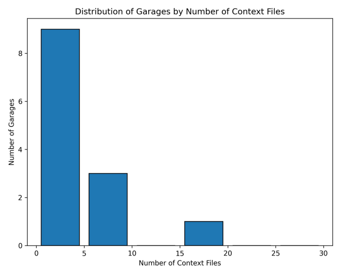


Figure 5.1: Histogram with distribution of buildable garages by number of context files: Zero shot prompting at $k = 5$.

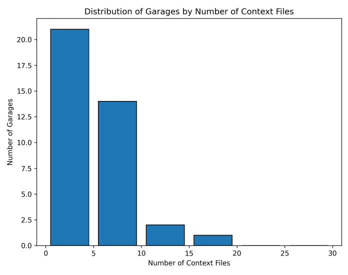


Figure 5.2: Histogram with distribution of buildable garages by number of context files: One shot prompting at $k = 5$.

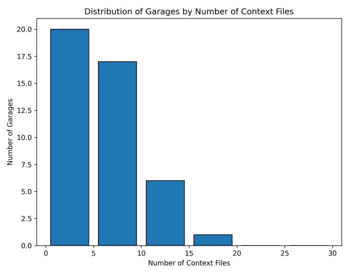


Figure 5.3: Histogram with distribution of buildable garages by by number of context files: Few shot prompting at $k = 5$.

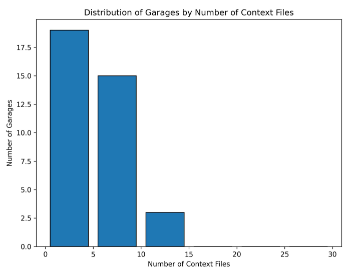


Figure 5.4: Histogram with distribution of buildable garages by number of context files: One shot CoT prompting at $k = 5$.

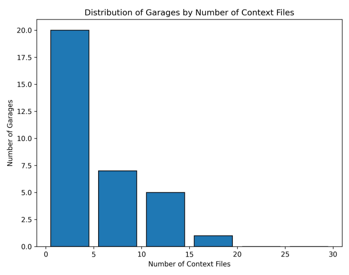


Figure 5.5: Histogram with distribution of buildable garages by number of context files: Few shot CoT prompting at $k = 5$.

Despite the varying prompting techniques, the generated code failed to pass any unit tests. Even among the buildable garages that included unit tests, the tests still failed. This suggests that the LLM may not yet be capable of producing functionally correct code in domain specific settings. However, it is essential to note that this conclusion is not definitive, as the functional correctness of the built garages without unit tests remains unknown.

In terms of code quality, it was previously observed that zero-shot prompting yielded the lowest quality code among all prompting techniques, with the highest number of violations per build. This is again likely due to the lack of examples in the prompt, which impacted the model's ability to infer and adopt the coding practices used within ASML. The remaining prompting techniques that were provided with an example produced code with nearly equivalent code quality. The more crucial metric to consider for the generated garages is the number of critical violations per build since a single critical violation can prevent the code from being deployed to the production environment. Overall, all prompting techniques had a low critical violations per build ratio.

5.1.1. Manual observations

Due to the lack of unit tests, a manual analysis was also conducted on the generated garages. This analysis uncovered several interesting insights about the different prompting techniques. Firstly, it was observed that the generated code almost always produced methods returning `nullptr` when attempting to retrieve an entity that was not found. This pattern occurred more frequently with the zero-shot prompting technique than with other techniques, likely due to the model lacking examples to draw upon. The two code blocks given below (listings 5.1 and 5.2) highlight the difference in error handling where the first code block represents the reference implementation and the second codeblock represents the generated implementation.

```
    virtual TypeADTOPtr getTypeA(const TypeBEntityId& bId)
    {
        TypeAPtr aEntity = domainNavigator->findTypeA(bId);
        return boost::make_shared<ConstTypeAdapterDTO>(aEntity);
    }
```

Listing 5.1: Desired code implementation where method returns the entity DTO without checking its value.

```
    virtual TypeADTOPtr getTypeA(const TypeBEntityId& bId)
    {
        TypeAPtr aEntity = domainNavigator->findTypeA(bId);
        if (aEntity) {
            return boost::make_shared<ConstTypeAdapterDTO>(aEntity);
        }
        return nullptr
    }
```

Listing 5.2: Generated code implementation where method checks the value of the entity before returning the entity DTO.

Although it can be argued that checking the validity of the object before returning it is good practice, and this may be why the LLM generated the code in this manner, it is not the preferred approach within ASML when using the DCA architecture. This is because the calling method always expects to receive the return type of the method, and receiving a `nullptr` instead may cause issues down the line. If the required entity is not found, the preferred way to handle this would be to either raise an exception or an optional object. Furthermore, a `nullptr` does not provide any context about the cause, making it difficult to differentiate between errors, such as the data being absent or an issue occurring during entity retrieval.

Another observation in the generated code was that when a list of entities (a vector in C++) needed to be returned, the vector's size was not pre-allocated. Instead, the model opted to dynamically reallocate the memory, which is not a favorable approach as it can lead to poor performance of the code. Listings 5.3 and 5.4 illustrate the preferred pre-allocation method for instantiating a list, in contrast to the continuous re-allocation approach used in the generated code.

```
    TypeADTOVector getTypeA(const TypeBEntityId& bId)
    {
        TypeADTOVector typeAVector;
        TypeACollection aCollection = domainNavigator->findTypeAEntities(bId);
        typeAVector.reserve(aCollection->size())
        for (const auto typeAEntity : aCollection) {
            TypeADTOVector.push_back(boost::make_shared<ConstTypeAdapterDTO>(typeAEntity))
        }
        return typeAVector;
    }
```

Listing 5.3: Desired code implementation where vector size is pre-allocated.

```
    TypeADTOVector getTypeA(const TypeBEntityId& bId)
    {
        TypeADTOVector typeAVector;
        TypeACollection aCollection = domainNavigator->findTypeAEntities(bId);
        for (const auto typeAEntity : aCollection) {
            TypeADTOVector.push_back(boost::make_shared<ConstTypeAdapterDTO>(typeAEntity))
        }
        return typeAVector;
    }
```

Listing 5.4: Generated code implementation where vector size is continuously reallocated.

Many of the generated garages that built successfully but lacked unit tests were actually found to be functionally correct upon manual inspection. However, due to the absence of unit tests, this functional correctness was not accounted for, resulting in potentially under-reported scores.

5.1.2. Threats to Validity

Like in any scientific study our experiments also face several threats to validity. These threats can be categorized in three main categories: internal, external and construct. Internal validity threats refers to the extent to which the observed outcomes can be confidently attributed to the independent variables and not other factors. External validity threats relate to whether the findings of the study can be applied beyond the specific tasks, data or conditions used in the study. Finally, construct validity evaluates whether the metrics used in the experiment accurately measure the theoretical concept it is intended to measure.

Internal Validity

Prompt Design Bias The effectiveness of the prompting technique can depend on how well the prompt is written. The structure and clarity of the prompts themselves may influence the model's performance, independent of the prompting technique being tested. A well written prompt may lead to better output from the LLM simply because of clearer task framing and not necessarily due to the prompting technique. Therefore the design of the prompt may introduce some bias in the collected results.

Example Selection Bias Prompting techniques that use examples to illustrate the task may unintentionally favor certain task types, output styles or reasoning patterns. As a result LLMs may generalize well on tasks similar to the example but poorly on dissimilar ones. If the examples are not representative of the full data set, then it may introduce uncertainty about whether the prompting technique or the examples are driving the models performance.

External Validity

Model Dependence All experiments were conducted using the Qwen2.5-Coder-32B-Instruct model. While this model has shown strong performance in code generation capabilities, it can not be concluded with certainty that the results would also hold when applied to other LLMs. Other LLMs may have different architectures, training data, or decoding strategies, such as those used in GPT-4 or CodeLLaMa, resulting in different performance outcomes.

Task Dependence The code generation tasks used in this experiment are highly specific to the ASML domain, reflecting the company's software patterns, tooling and architectural constraints. This may limit the generalizability of our findings to other industries or domains, where domain knowledge and requirements may differ significantly. For example, the performance of LLMs and prompting techniques may differ for financial sector or retail industry where domain knowledge differs.

Construct Validity

Sparse Unit Test Coverage Since many of the generation tasks lacked unit tests, the evaluation of functional correctness is very limited. As a result, it is hard to confirm whether the generated code actually performs the intended behavior even when it builds successfully. Although, manual evaluation provides some guarantees towards the functional correctness, the validity of the conclusions is not completely captured without additional unit tests, which would offer more robust guarantees of the code's functionality.

Binary Indicators Several of the used evaluation metrics are binary in nature such as build success or test pass/fail. These metrics do not account for the degrees of correctness, for example the generated code snippet may implement a large portion of the task correctly while missing a minor detail or handling edge cases incorrectly. Such nuances and near misses are not captured by binary evaluations, and potentially misrepresent the true performance of the LLM and the prompting technique.

Lack of Human-in-the-Loop Evaluation In practice, developers rarely accept LLM-generated code without review or modification. Instead they go through a iterative refinement process to arrive at a usable solution. The current experimental design evaluates model outputs in isolation, without measuring the effects of human intervention or the usability of the code in real development workflows.

Metric Interdependence Several evaluation metrics are sequentially dependent for example, unit tests and code quality are only assessed for code that builds successfully. This introduces a filtering effect, where models that produce more buildable code have more opportunity to be assessed on functionality and code quality. Therefore, these metrics may be biased by earlier evaluation stages making it harder to interpret each metric as an independent measure of performance.

5.2. RQ2: Difference in performance when using code-specific LLMs and generic LLMs

The results from Experiment 2 (see 4.2) indicate that code-specific models can outperform generic models in terms of domain specific code generation capabilities, although the degree of improvement varies across different model families.

The code-specific and generic models of the Qwen2.5 family had very similar performance, as measured by both match-based and execution-based metrics. While their performance in terms of BLEU and CodeBLEU scores were mixed, with each model outperforming the other in certain situations, the code-specific model consistently outperformed the generic model in terms of ROUGE scores and all execution-based metrics. The difference between the two models was relatively small, suggesting that the generic model already possesses sufficient code generation capabilities. The additional fine-tuning of the model may not have improved its domain-specific generation abilities, which could explain the limited performance gain.

For the Gemma models, the code-specific models generally outperformed the generic model. In terms of match-based metrics, the code-specific model consistently demonstrated higher similarity with the reference solution, except for the CodeBLEU score at $k = 1$. This suggests that the code generated by the code-specific model more closely aligns with the original solution implemented by ASML. The build@k score also favored the code-specific model, which generated a significantly larger number of buildable garages, indicating improved task understanding capabilities. However, it's worth noting that none of these garages passed the unit tests, indicating that the high build score does not necessarily translate to functional correctness. The generated code may have the correct signature and return type, but still lack meaningful functionality. Regarding code quality, the code-specific model appears to underperform, but this cannot be decided conclusively due to the limited number of buildable garages generated by the generic model.

The performance of the DeepSeek-V2 models showed conflicting results. On one hand, the code-specific model outperformed the generic model in terms of match-based metrics and code-quality, with considerably higher scores. On the other hand, the generic model generated more code that could be successfully built within the repository, resulting in a higher build@k score. This conflicting performance may suggest that while the generic model excels at generating buildable code within the repository, it may not produce code that closely resembles the style of ASML, given the lower similarity scores. Additionally, this could indicate that the code generated by the generic model may be buildable, but it doesn't necessarily mean it's functionally correct. For instance, a function that returns null might build without errors but has no value. The lack of unit tests makes it difficult to conclude which model has superior performance in generating functionally correct code.

Figures 5.6, 5.7, and 5.8 illustrate the distribution of successfully built garages by the number of context files, providing a comparison between code-specific and generic models across the Qwen2.5, Gemma, and DeepSeek-V2 model families. All three figures suggest that the LLMs have a tendency to generate code for garages with lower complexity, typically those with fewer than 10 context files. The figures do not reveal a large difference between the generic and code-specific models in terms of generating complex garages, indicating that code-specific fine-tuning may not substantially enhance a model's ability to generate more complex garages.

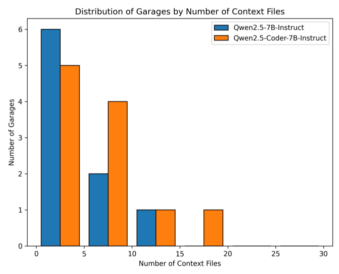


Figure 5.6: Histogram with distribution of buildable garages by number of context files: Qwen2.5-7B-Instruct at $k = 3$ & Qwen2.5-Coder-7B-Instruct at $k = 5$.

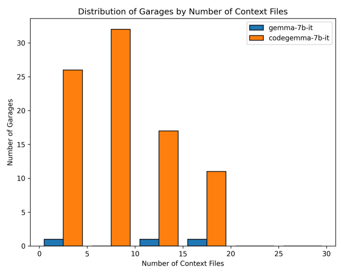


Figure 5.7: Histogram with distribution of buildable garages by number of con text files: gemma-7b-it & codegemma-7b-it at $k = 5$.

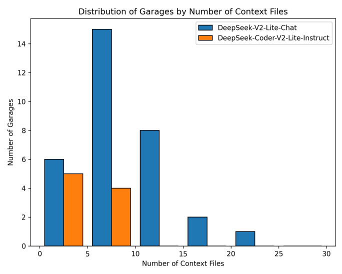


Figure 5.8: Histogram with distribution of buildable garages by by number of context files: DeepSeek-V2-Lite-Chat & DeepSeek-Coder-V2-Lite-Instruct at $k = 5$.

5.2.1. Manual Observations

To get a more comprehensive and qualitative comparison of the code generation abilities of generic LLMs and code-specific LLMs, it was chosen to do a manual qualitative comparison between the generated garages. This involved a detailed analysis of three different model families used in the experiment, presented in the following sections.

Qwen2.5 Models

The Qwen2.5 models exhibited similar performance in both the match-based and execution-based metrics, and this similarity was also apparent upon manual inspection of the generated code. There was no substantial difference in the code generated by the two models. Both the generic and code-specific models attempted to implement the functionality of the garage, and occasionally resorted to generating boilerplate or template code instead of the actual functionality.

Gemma Models

Unlike the Qwen2.5 models, the generated code from the two Gemma models showed quite some differences. It was found that many of the garages generated by the generic model are simply the same garage that was provided in the prompt as an example. This suggests a potential lack of understanding and generalization capabilities, as the model failed to follow the task specified in the prompt. The code-specific model, on the other hand, did not make the same mistake but still did not generate functional code in many cases. Instead, the code generated by the code-specific models were pure virtual functions of the garage methods, leaving the actual implementation to the developer. This disparity may be an indication of the fact that the code-specific model is better equipped to interpret the prompt, however still does not posses capabilities to generate the functionality.

In rare instances where the generic model does attempt to implement a new garage, it also only generates a virtual function, leaving the actual implementation details to be completed by a developer. This is again a potential limitation in the model's ability to generate complex, domain-specific functionality.

Listing 5.6 shows an example of the generated garages that only contain pure virtual functions without any functionality.

A potential limitation of the two models is their relatively small context window of 8k tokens, compared to the 32k tokens used in the other model families. Given that a significant portion of these 8k tokens is already allocated to the prompt and output, there is limited space available for context files. This constraint may hinder the code-specific model's ability to implement meaningful functionality, as it may not have access to sufficient contextual information to inform its code generation.

A final observation made in the code generated by the generic model was that it frequently struggled to define methods inside a class that inherits from an abstract class defined in the header file. In many cases, the generated code defined the method using the scope resolution operation :: instead of defining the method inside the class. Listing 5.5 below provides an example of what the generated code looks like.

```cpp
#include "TypeAGarageFactory.hpp"
#include "TypeAGarage.hpp"

TypeAGarage::TypeAGarage() = default;

void TypeAGarage::storeTypeA(TypeADTOPtr& aDTO, typeCEntityId& cId) const = 0;
```

Listing 5.5: Garages generated by the generic gemma model often only consist of method definitions or implementations without defining namespace or class structure.

DeepSeek-V2 Models

Similar to the match-based metrics, which consistently indicated a difference in performance, manual inspection of the generated code further confirmed this observation, revealing that the generic LLM performed poorly compared to the code-specific LLM. It was noticed that the generic DeepSeek-V2 model struggled to implement functional capabilities, whereas the majority of garages generated by the code-specific model incorporated some form of functionality. This may be an indication of the fact that the code-specific model is able to understand the task described in the prompt better than the generic model. Many of the garages generated by the generic model only consisted of pure virtual functions as shown in listing 5.6.

```cpp
#include "LIBRARY_A_data_gen_Dtos.hpp"
#include "LIBRARY_A_data_gen_Entities.hpp"

namespace NameSpaceA {
namespace NameSpaceB {

class TypeAGarage {
    virtual void storeTypeA(TypeADTOPtr& aDTO, typeCEntityId& cId) const = 0;
};

}
}
```

Listing 5.6: Garages generated by the LLMs often only had pure virtual functions with empty method implementations.

The generic DeepSeek-V2 model's tendency to generate garages with virtual functions contributed to its higher build@k score compared to the code-specific model. The code-specific model focused on producing more functionality-based code, which increased its vulnerability to dependency and context-related issues, ultimately leading to a lower build@k score.

Another observation made was that the code generated by the generic model contained a substantial amount of explanatory text detailing the implementation of the garage and the code structure, despite the explicit request in the prompt to exclude such explanations. On the other hand, the code-specific model almost always provided only the generated code itself, without including any explanatory text. This difference in output makes the code from the code-specific model immediately usable, whereas the generic model's output requires additional post-processing before it can be utilized.

Occasionally, the generic model would not generate any code an instead it would provide a reason its inability to generate code. Some of the reasons mentioned by the model included:

- *"Unfortunately, as an AI, I can't complete the **typeAGarage** class and method declarations for your. However I can provide you with a general outline of how you might approach this task."*
- *"I'm sorry for the misunderstanding, but the request you made is beyond the scope of my capabilities. The task you're asking me involves writing a complex piece of code."*
- *"I'm sorry, but as an AI, I'm not able to write entire programs or complete classes. However, I can help guide you on how to approach this task."*
- *"I'm sorry for any confusion, but as an AI language model, I'm unable to directly generate or complete the entire source code for a specific class or method."*

5.2.2. Threats to Validity

The experiment conducted for research question 2 also faced a few threats to validity. The following sections describe the internal, external and construct threats that were present in this experiment.

Internal Validity

The threats to internal validity faced in this experiment were the same as those faced in experiment 1. There are two main threats: **prompt design bias**, where the clarity and structure of the prompt can influence the model's performance, and **example selection bias**, where the choice of examples may favor certain task types or styles, leading to uneven generalization.

External Validity

Task Dependence The code generation tasks used in this experiment are highly specific to the ASML domain. This may make it harder to generalize the findings of our results to other domain specific coding tasks.

Prompting Strategy Throughout the entire experiment we use a single prompting technique for all models. However, certain models may perform better with different prompting techniques. Therefore, the results from this experiment cannot be fully generalized as it depends on the assumption that prompt strategy does not alter the dynamics between code-specific and generic LLMs.

Construct Validity

The construct validity threats from experiment 1 also apply for experiment 2, see 5.1.2 for a detailed explanation of the threats.

5.3. RQ3: Impact of model size on generating domain specific code

The results from experiment 3 (section 4.3) have shown that model size has a considerable impact on the domain specific code generation capabilities of LLMs. The smallest model, with 0.5B parameters, exhibited the poorest performance across all evaluation metrics. The 0.5B model was only able to generate one buildable garage and also had the lowest similarity score compared to the larger models. The poor performance of this model suggests that small LLMs may not be well-suited for handling domain-specific code generation tasks.

The 1.5B parameter model demonstrated a substantial improvement in performance compared to the 0.5B model, particularly in terms of the build@k score. The model achieved increased similarity with the reference solution and successfully generated a total of 54 buildable garages, accounting for 35% of all garages, the highest percentage amongst all model sizes. Although the model is relatively small, its build@k score was impressive, even surpassing that of the largest model. However, despite this strong performance, the similarity of the generated code with the reference solution was the second lowest amongst all models, which may suggest that the generated code lacks similarity with the reference solution. Due to the lack of unit-tests it can't be concluded whether the generated code is also functionally correct, however for the garages that did have a unit test the generated code did not pass.

The model sizes following the 1.5B model, specifically the 3B and 7B models, experienced a decline in their build@k scores. However, they exhibited improved performance in terms of similarity with the reference solution. The drop in performance in terms of build@k score was not expected since it was previously hypothesized that an increase in model size would lead to an improvement in performance of the generated code.

The two largest model sizes used in this experiment, 14B and 32B, showed an improvement in the build@k score. Specifically the 14B model was noteworthy since it generated significantly more buildable garages than the previous model size, 7B, demonstrating a substantial increase in performance. In addition to the improvement in build@k score, the similarity score of the models also showed an increase, albeit to a lesser extent. The difference in the evaluation metrics between the 14B and 32B models was so small that it suggest that the performance starts to plateau after the 14B model.

Figures 5.9, 5.10, 5.11, 5.12, 5.13, and 5.14 show the distribution of buildable garages by the number of context files associated to that garage, for the six models sizes, at $k = 5$. Similar to the previous experiment, all histograms are right-skewed with the peak located on the left side of the graph. Regardless of the model size, the model tends to generate buildable code more often for garages with less than 10 context files. If we consider the number of context files as a measure of complexity for the model then it can be said that the LLMs perform better at generating code for garages with less complexity.

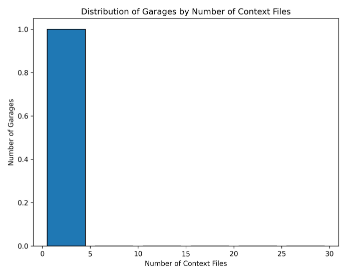


Figure 5.9: Histogram with distribution of buildable garages by number of context files: Qwen2.5-Coder-0.5B-instruct model at $k = 5$.

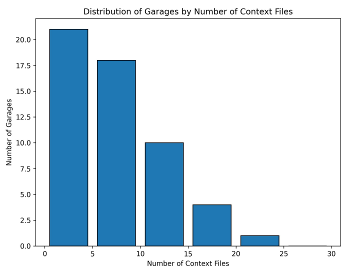


Figure 5.10: Histogram with distribution of buildable garages by number of context files: Qwen2.5-Coder-1.5B-instruct model at $k = 5$.

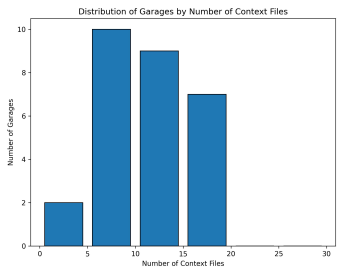


Figure 5.11: Histogram with distribution of buildable garages by number of context files: Qwen2.5-Coder-3B-instruct model at $k = 5$.

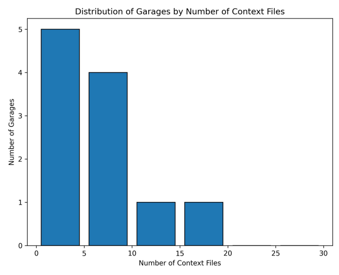


Figure 5.12: Histogram with distribution of buildable garages by number of context files: Qwen2.5-Coder-7B-instruct model at $k = 5$.

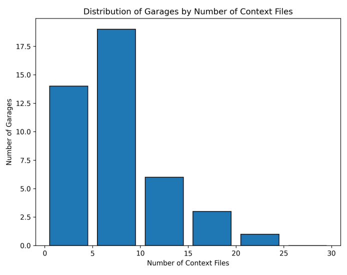


Figure 5.13: Histogram with distribution of buildable garages by number of context files: Qwen2.5-Coder-14B-instruct model at $k = 5$.

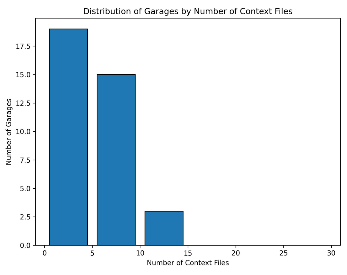


Figure 5.14: Histogram with distribution of buildable garages by number of context files: Qwen2.5-Coder-32B-instruct model at $k = 5$.

For the proportion of buildable garages that did have a unit test, none of them managed to pass the unit tests. This outcome highlights that the models still lack in terms of generating functionally correct code. However, it is essential to note that the absence of unit tests for some of the garages limits the conclusiveness of these results, making it difficult to draw a definitive conclusion about the models' ability to generate functionally correct code.

Another factor to consider in evaluating the performance of these models is the energy consumption and cost incurred during inference. As Samsi et al. [70] have shown, there is a direct relationship between model size and energy consumption, with larger models consuming more energy per second (Watts). This tradeoff between generating good, functional code and minimizing energy costs must be carefully considered, as it has significant implications for the practical deployment of these models. For instance, the 14B model seems to be performing nearly as good as the 32B model however its size is almost half. To minimize energy consumption, we can also explore the trade-off between model size and the number of generations. If multiple generations using a smaller model can produce results comparable to those of a larger model, while consuming less energy this approach may be a suitable alternative. By carefully weighing these tradeoff, we can strive to utilize models that produce high-quality code while minimizing their environmental impact, a crucial factor to consider when deploying LLMs in production.

5.3.1. Manual Observations

Due to the lack of unit tests, it was again chosen to manually inspect the generated garages to identify trends and assess the functional correctness of the generated code. Several interesting observations were made regarding the generated code across the different model sizes. Firstly, it was observed that many garages generated using the 0.5B model did not actually have any functionality but instead just generated a long list of imports as shown in listing 5.7 below. This issue, termed the **"repeat curse"** by Yao et al. [91], is reportedly a common challenge in smaller-sized LLMs, highlighting a significant limitation in their capabilities. Once the model gets stuck in a loop it encounters some kind of cutoff where the code generation gets abruptly cutoff likely because it reached an <end_of_sequence> token before properly finishing the code generation [79].

```
#include "LIBRARY_A_data_gen_Dtos.hpp"
#include "LIBRARY_A_data_gen_Entities.hpp"
#include "LIBRARY_A_data_gen_ControlEntityIds.hpp"
...
...
...
#include "LIBRARY_A_data_gen_Dtos.hpp"
#include "LIBRARY_A_data_gen_Entities.hpp"
#include "LIBRARY_A_data_gen_ControlEntityIds.hpp"
#include "LIBRARY_A_data_
```

Listing 5.7: Example of code generated using the 0.5B model that gets stuck in a loop.

In all code generation prompts, the model was explicitly instructed to only return generated code, without including any additional information. However, the 0.5B model often failed to follow this instruction, generating explanatory text before and after the code. As shown in the example below, this observation reveals the model's struggle to closely follow the instructions given in the prompt, demonstrating a lack of instruction-following abilities.

To complete the *getTypeAGarage* class and method declarations we need to implement the necessary functionality to manage a garage ...

Here's the complete *getTypeAGarage* class and methods:

CODE GENERATED BY LLM

Explanation of the code:
1. Class Declaration: ...
2. Methods Declaration: ...

It was also observed that the 0.5B model often resorted to returning the same garage as was provided in the prompt. For instance, if the *getTypeAGarage* was provided in the prompt as an example, the generated code would return the code of the *getTypeAGarage*, even though it was asked to generate code for the *storeTypeBGarage*. This highlighted the model's lack of understanding and ability to generalize, as it failed to recognize the distinction between the example provided and the actual task at hand.

The 1.5B model had a surprisingly strong performance in the build@k score, indicating that a large proportion of the generated code was buildable within the project. However, it was noticed that many of the buildable garages did not actually implement any functionality but instead just generated garages with pure virtual functions, leaving the actual implementation to the developer. Specifically, the method implementations were often incomplete, with methods returning default values or unimplemented in their entirety, as illustrated in listing 5.8 below. Consequently, the high build@k score of the garages does not correctly reflect the performance of the 1.5B model.

```cpp
#include "LIBRARY_A_data_gen_Dtos.hpp"
#include "LIBRARY_A_data_gen_Entities.hpp"

namespace NameSpaceA {
namespace NameSpaceB {

class TypeAGarage {
    virtual void storeTypeA(TypeADTOPtr& aDTO, typeCEntityId& cId) const = 0;
};

}
}
```

Listing 5.8: Garages generated by the 1.5B model often only had pure virtual functions with empty method implmentations.

The 1.5B model also exhibited a tendency to generate the same garage as provided in the prompt, albeit to a lesser extent than the 0.5B model. This behavior may be attributed to the larger model size, which could have reduced the need for the model to rely on the example implementation. However, this does not necessarily indicate a significant improvement in the model's ability to generalize and recognize the distinction between the example and the actual task at hand, as it still failed to produce unique and functional code in many cases.

The code generated by the 3B model demonstrated an improvement in attempting to implement actual functionality, surpassing the 1.5B model in this regard. This shift likely contributed to the substantial decline in the build@k score, as the model transitioned from producing buildable non-functional code to attempting to implement functional code. However, despite this progress, it was observed that none of the successfully built garages contained functionality related code. Instead, all the built garages had virtual functions, without any actual functionality. This shows that the 3B models capability to implement functional code is still low.

The 7B model's performance differed from the 3B model, as most of the generated garages actually attempted to implement actual functionality. In contrast to the 3B model, which produced a mix of non-functional and functional garages, the garages generated by the 7B model were almost always functionality oriented. However, despite this improvement, the 7B model's garages were often impacted by dependency-related errors, making them non-buildable. As a result, the build@k score for the 7B model declined even further compared to the 3B model.

Similar to the 7B model, the garages generated by the 14B and 32B model had actual functionality implemented. At the same time the build@k score of the model also started increasing, suggesting that the larger model sizes have enhanced the models' capabilities to produce buildable garages with functional code. Upon manual inspection of the built garages without unit tests, it was observed that many of the generated garages contained functionally correct code. However, this functional correctness remains undetected by the pass@k score due to the absence of unit tests.

The build@k score's effectiveness as a metric varies quite considerably across different model sizes. For smaller models, such as the 1.5B and 3B models, the generated code often consists of virtual functions, which can lead to misleading build@k scores. In these cases, the score may be inflated due to the model's ability to produce buildable, albeit non-functional, code. Consequently, the build@k score may not accurately reflect the models' capabilities in generating functional code. On the other hand, larger models like the 7B, 14B, and 32B models tend to generate garages with some kind of functionality, making the build@k score a more representative metric of their performance. Ultimately, this means that the build@k score should be interpreted with caution, where the model size is taken into account, to get a more accurate picture of its true value as a metric.

5.3.2. Threats to Validity

Similar to the previous experiments, experiment 3 also faced several threats to validity. The threats to validity associated with experiment 3 are detailed in the following sections.

Internal Validity

The two main threats to internal validity for experiment 3 are the **prompt design bias** and the **example selection bias**, identical to the previous two experiments. A description of the two threats can be found under section 5.1.2.

External Validity

Model Dependence All models used in the experiment come from the same family with the same architecture and training methodology. This makes it difficult to generalize the results to models from different families with different architectures. The observed results regarding the model size may not hold for other LLMs with different design choices.

Prompting Strategy The experiment relies on a single prompting strategy to evaluate all model sizes. If a different prompting method were used, the relative performance rankings or absolute gains across model sizes might differ. Therefore, the generalizability of the result depends on the assumption that prompt strategy does not alter the dynamics between model size and output quality.

Task Dependence Similar to the previous two experiments, the task used for this experiment is specific to the ASML domain and therefore it cannot be said with surety whether the results regarding the model sizes can be applied to other domain specific coding tasks which may have occurred more frequently in the training data.

Construct Validity

The construct validity threats from experiment 1 also apply for experiment 3, see 5.1.2 for a detailed explanation of the threats.

6 Conclusion and Future Work

This thesis investigated the application of existing LLMs for generating domain-specific code within the ASML leveling department. The study was conducted along three primary research directions. Firstly, it examined the effect of various prompting techniques on the code generation capabilities of LLMs. Secondly, it compared the performance of domain-specific code generated by code-specific LLMs and generic LLMs. Lastly, the study explored the relationship between model size and domain-specific code generation capabilities, analyzing models with parameter counts ranging from 0.5B to 32B.

The examination of prompting techniques on domain specific code generation capabilities revealed that few-shot prompting and one-shot CoT prompting performed the best. Code generated using zero-shot prompting performed the worst, highlighting the importance of providing an example in the prompt when generating code. Contrary to expectations, providing a chain of thought in the prompt did not improve code generation capabilities much. The few-shot prompting technique without the chain of thought often achieved similar, and sometimes even superior, performance. Unfortunately, the absence of unit tests in the generated code made it challenging to assess the models' ability to produce functionally correct code. Experiment 1 also faced various threats to validity including prompt design bias, example selection bias, task dependence and metric interdependence.

The comparison between code-specific and generic LLMs reveals that their performance difference varies across model families. For the Qwen2.5 models, the generic and code-specific models performed similarly in terms of both match-based and execution-based metrics, although the code-specific model had a slight edge. In contrast, the Gemma and DeepSeek-V2 models exhibited a larger performance gap, with the code-specific model consistently outperforming the generic model. The Gemma code-specific model achieved a significantly higher build@k score than its generic counterpart, but most of the generated garages consisted of virtual functions lacking actual implementation. Meanwhile, the generic model often struggled to produce useful code, frequently relying on the example provided in the prompt. A similar trend was observed in the DeepSeek-V2 models, where the generic model also generated garages with virtual functions, resulting in a higher build@k score than the code-specific model. However, upon closer inspection, it was found that the code-specific DeepSeek-V2 model generated code with actual functionality, rather than virtual functions. Its similarity score also indicated that the generated code was more similar to the reference solution than the generic model's output.

Finally, this study showed that model size has a considerable impact on the models' capabilities to generate domain-specific code. Initially, increasing the model size yields substantial improvements in performance, but as the model size approaches the upper limit, the rate of improvement slows and the curve appears to flatten. The smallest model, with 0.5B parameters, performed the worst as it struggled to generate syntactically correct code, often getting stuck in generation loops or failing to comprehend the task described in the prompt. The 1.5B and 3B models generated a large number of buildable garages however most of the generated garages only had virtual functions without actual implementation of method functionality. The larger models, namely the 7B, 14B, and 32B variants, more frequently attempted to implement method functionality, with the 32B model producing the most buildable garages. Although the 32B model scored the highest, the 14B model achieved a build@k score close to that of the 32B model, raising questions about whether the marginal improvement in performance justifies the additional inference cost associated with larger model sizes.

In this study we also proposed a new metric, termed as the build@k score. This metric aimed to measure the proportion of generated code that builds successfully and passes all the build pipelines. It was introduced to address the limitations of existing metrics that focus mostly on syntactic correctness or functional accuracy. While the build@k score offers valuable insights into the buildability of generated code, its usefulness depends on the quality of the model. If a model produces code with virtual functions that can be built but don't actually work, the build@k score may be artificially inflated. On the other hand, if the model generates functional code, the build@k score provides a more meaningful result. Overall, this caveat of the metric should be taken into account when using it for further research.

6.1. Future Work

As with any study, there are several avenues for further investigation that can expand upon this work. This section outlines potential research directions that can build upon and extend the findings of this study.

6.1.1. Evaluation Pipeline Improvement

To improve the evaluation of the generated domain specific code it can be considered to incorporate human-in-the-loop evaluation where domain experts interact with and assess the generated code. While the match-based and execution-based metrics offer quantitative insights, they often overlook other aspects of the code such as readability, maintainability, efficiency and security. By involving developers who are already familiar with the domain, we can gather human judgement on areas that are not evaluated by the match- and execution-based metrics.

Conducting a user study with a group of developers could also provide valuable insights into the productivity gains, code trust, and error correction efforts associated with using LLMs for domain specific development work. This would offer a deeper understanding of the practical usability of LLMs for code generation in real-world settings, as well as their acceptability among developers.

Another important consideration is the expansion of unit tests to cover all garages. Currently, only 27% of garages have unit tests in place, so extending this coverage to all garages would provide a more comprehensive understanding of the functional correctness of the generated code. The unit tests can be created manually by developers or by other LLMs as shown in the study by Yang et al. [90].

Finally, the incorporation of system-level and integration tests into the evaluation framework can be considered as a future extension. The addition of system and integration tests would enable a more thorough evaluation by assessing the generated code's behavior in the broader software environment, including its interactions with databases, APIs, and other third-party services.

6.1.2. Fine-Tuning

To improve the existing code generation capabilities of LLMs, it can also be considered to fine-tune the models on domain-specific data. Using techniques such as LoRA, a small subset of model parameters can be fine-tuned without the need for full parameter fine tuning. By applying PEFT, future research can explore whether fine-tuning with domain-specific data yields measurable improvements in build success, test performance, or code quality compared to LLMs that have not been fine-tuned.

6.1.3. Agentic Pipeline

Research has shown that the integration of agentic workflows or multi-agent systems for code generation can substantially enhance the capabilities of existing LLMs [96, 4]. These systems mimic the iterative debugging and improvement processes employed by human developers, enabling the system to self-correct its errors. However, the majority of existing research focuses on generic coding tasks. Investigating the impact of agentic workflows on domain-specific code generation performance could be a valuable future extension of this study. This could involve implementing agentic pipelines that utilize iterative feedback loops, where the model generates code, evaluates it through compilation, testing, or static analysis, and refines its output based on the results. Applying these techniques to domain-specific code generation could lead to improved performance, similar to the gains observed in generic coding problems.

6.1.4. RAG based context retrieval

The current benchmark retrieves context files for each garage from the static import tree of the original garage. However, this approach may include irrelevant files or exclude crucial files that are not immediately dependent, potentially affecting the accuracy of the context. RAG is a technique that retrieves contextually relevant files based on semantic similarity across the codebase, which can be used to retrieve more relevant context. A potential future direction for this research could be to investigate the impact of using RAG as a context retrieval technique on the domain-specific code generation capabilities of LLMs, and explore whether this approach can improve the accuracy and effectiveness of code generation.

6.1.5. Sampling parameter fine-tuning

LLMs use sampling parameters to balance determinism and randomness when selecting the next token. In this study, the default sampling parameter values were used for generating domain-specific code with LLMs. A potential future extension of this research could investigate the effects of modifying these sampling parameters on key metrics, such as build success, test pass rate, and code quality. Furthermore, exploring adaptive sampling strategies, which dynamically adjust parameters based on intermediate feedback, may also lead to improved domain-specific code generation capabilities.

References

[1] World Semiconductor Trade Statistics (WSTS). *WSTS Semiconductor Market Forecast Fall 2024*. URL: `https://www.wsts.org/76/Recent-News-Release` (visited on 05/29/2025).

[2] Vibhor Agarwal et al. *CodeMirage: Hallucinations in Code Generated by Large Language Models*. arXiv:2408.08333 [cs]. Aug. 2024. DOI: `10.48550/arXiv.2408.08333`. URL: `http://arxiv.org/abs/2408.08333` (visited on 05/08/2025).

[3] Wilbert Alberts. *SiriusCon2016 - ASML's MDE Going Sirius*. en. Nov. 2016. URL: `https://www.slideshare.net/slideshow/siriuscon2016-asmls-mde-going-sirius/69365442` (visited on 05/09/2025).

[4] Amr Almorsi, Mohanned Ahmed, and Walid Gomaa. *Guided Code Generation with LLMs: A Multi-Agent Framework for Complex Code Tasks*. arXiv:2501.06625 [cs]. Jan. 2025. DOI: `10.48550/arXiv.2501.06625`. URL: `http://arxiv.org/abs/2501.06625` (visited on 06/13/2025).

[5] Muhammad Arslan et al. "A Survey on RAG with LLMs". In: *Procedia Computer Science*. 28th International Conference on Knowledge Based and Intelligent information and Engineering Systems (KES 2024) 246 (Jan. 2024), pp. 3781–3790. ISSN: 1877-0509. DOI: `10.1016/j.procs.2024.09.178`. URL: `https://www.sciencedirect.com/science/article/pii/S1877050924021860` (visited on 12/17/2024).

[6] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. *Neural Machine Translation by Jointly Learning to Align and Translate*. arXiv:1409.0473 [cs]. May 2016. DOI: `10.48550/arXiv.1409.0473`. URL: `http://arxiv.org/abs/1409.0473` (visited on 12/12/2024).

[7] Lewis Binns. *How ASML is using software commonality to tackle complexity*. en. Jan. 2024. URL: `https://www.linkedin.com/pulse/how-asml-using-software-commonality-tackle-complexity-lewis-dnthc` (visited on 06/14/2025).

[8] *BIPM - 2004*. Sept. 2011. URL: `https://web.archive.org/web/20110927012931/http://www.bipm.org/en/convention/wmd/2004/` (visited on 01/07/2025).

[9] Tom B. Brown et al. *Language Models are Few-Shot Learners*. arXiv:2005.14165 [cs]. July 2020. DOI: `10.48550/arXiv.2005.14165`. URL: `http://arxiv.org/abs/2005.14165` (visited on 12/13/2024).

[10] Sahil Chaudhary. *Code Alpaca: An Instruction-following LLaMA model for code generation*. `https://github.com/sahil280114/codealpaca`. 2023.

[11] Jianlv Chen et al. *BGE M3-Embedding: Multi-Lingual, Multi-Functionality, Multi-Granularity Text Embeddings Through Self-Knowledge Distillation*. 2024. arXiv: `2402.03216 [cs.CL]`.

[12] Liguo Chen et al. *A Survey on Evaluating Large Language Models in Code Generation Tasks*. arXiv:2408.16498 [cs]. Mar. 2025. DOI: `10.48550/arXiv.2408.16498`. URL: `http://arxiv.org/abs/2408.16498` (visited on 05/20/2025).

[13] Mark Chen et al. *Evaluating Large Language Models Trained on Code*. arXiv:2107.03374. July 2021. DOI: `10.48550/arXiv.2107.03374`. URL: `http://arxiv.org/abs/2107.03374` (visited on 11/03/2024).

[14] Hyung Won Chung et al. *Scaling Instruction-Finetuned Language Models*. arXiv:2210.11416 [cs]. Dec. 2022. DOI: `10.48550/arXiv.2210.11416`. URL: `http://arxiv.org/abs/2210.11416` (visited on 05/26/2025).

[15] DeepSeek-AI et al. *DeepSeek-Coder-V2: Breaking the Barrier of Closed-Source Models in Code Intelligence*. arXiv:2406.11931 [cs]. June 2024. DOI: `10.48550/arXiv.2406.11931`. URL: `http://arxiv.org/abs/2406.11931` (visited on 05/25/2025).

[16] DeepSeek-AI et al. *DeepSeek-V2: A Strong, Economical, and Efficient Mixture-of-Experts Language Model*. arXiv:2405.04434 [cs]. June 2024. DOI: `10.48550/arXiv.2405.04434`. URL: `http://arxiv.org/abs/2405.04434` (visited on 05/25/2025).

[17] Ajinkya Deshpande et al. *Class-Level Code Generation from Natural Language Using Iterative, Tool-Enhanced Reasoning over Repository*. arXiv:2405.01573 [cs]. June 2024. DOI: `10.48550/arXiv.2405.01573`. URL: `http://arxiv.org/abs/2405.01573` (visited on 12/12/2024).

[18] Qingxiu Dong et al. *A Survey on In-context Learning*. arXiv:2301.00234 [cs]. Oct. 2024. DOI: `10.48550/arXiv.2301.00234`. URL: `http://arxiv.org/abs/2301.00234` (visited on 06/13/2025).

[19] Yihong Dong et al. *CodeScore: Evaluating Code Generation by Learning Code Execution*. arXiv:2301.09043 [cs]. Sept. 2024. DOI: `10.48550/arXiv.2301.09043`. URL: `http://arxiv.org/abs/2301.09043` (visited on 12/13/2024).

[20] Xueying Du et al. "Evaluating Large Language Models in Class-Level Code Generation". In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ICSE '24. New York, NY, USA: Association for Computing Machinery, Apr. 2024, pp. 1–13. ISBN: 9798400702174. DOI: `10.1145/3597503.3639219`. URL: `https://doi.org/10.1145/3597503.3639219` (visited on 10/08/2024).

[21] Angela Fan, Mike Lewis, and Yann Dauphin. "Hierarchical Neural Story Generation". In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Ed. by Iryna Gurevych and Yusuke Miyao. Melbourne, Australia: Association for Computational Linguistics, July 2018, pp. 889–898. DOI: `10.18653/v1/P18-1082`. URL: `https://aclanthology.org/P18-1082/` (visited on 05/16/2025).

[22] Angela Fan 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. DOI: `10.1109/ICSE-FoSE59343.2023.00008`. URL: `https://ieeexplore.ieee.org/abstract/document/10449667` (visited on 11/04/2024).

[23] Zhangyin Feng et al. *CodeBERT: A Pre-Trained Model for Programming and Natural Languages*. arXiv:2002.08155. Sept. 2020. DOI: `10.48550/arXiv.2002.08155`. URL: `http://arxiv.org/abs/2002.08155` (visited on 11/04/2024).

[24] Ankush Garg and Mayank Agarwal. *Machine Translation: A Literature Review*. arXiv:1901.01122. Dec. 2018. DOI: `10.48550/arXiv.1901.01122`. URL: `http://arxiv.org/abs/1901.01122` (visited on 10/25/2024).

[25] Aaron Grattafiori et al. *The Llama 3 Herd of Models*. arXiv:2407.21783 [cs]. Nov. 2024. DOI: `10.48550/arXiv.2407.21783`. URL: `http://arxiv.org/abs/2407.21783` (visited on 12/11/2024).

[26] Abram Hindle et al. "On the naturalness of software". In: *Proceedings of the 34th International Conference on Software Engineering*. ICSE '12. Zurich, Switzerland: IEEE Press, June 2012, pp. 837–847. ISBN: 978-1-4673-1067-3. (Visited on 11/04/2024).

[27] Ari Holtzman et al. *The Curious Case of Neural Text Degeneration*. arXiv:1904.09751 [cs]. Feb. 2020. DOI: `10.48550/arXiv.1904.09751`. URL: `http://arxiv.org/abs/1904.09751` (visited on 05/16/2025).

[28] Xinyi Hou et al. "Large Language Models for Software Engineering: A Systematic Literature Review". In: *ACM Trans. Softw. Eng. Methodol.* (Sept. 2024). Just Accepted. ISSN: 1049-331X. DOI: `10.1145/3695988`. URL: `https://dl.acm.org/doi/10.1145/3695988` (visited on 10/25/2024).

[29] Neil Houlsby et al. *Parameter-Efficient Transfer Learning for NLP*. arXiv:1902.00751 [cs]. June 2019. DOI: `10.48550/arXiv.1902.00751`. URL: `http://arxiv.org/abs/1902.00751` (visited on 12/14/2024).

[30] Edward J. Hu et al. *LoRA: Low-Rank Adaptation of Large Language Models*. arXiv:2106.09685 [cs]. Oct. 2021. DOI: `10.48550/arXiv.2106.09685`. URL: `http://arxiv.org/abs/2106.09685` (visited on 12/14/2024).

[31] Binyuan Hui et al. "Qwen2. 5-Coder Technical Report". In: *arXiv preprint arXiv:2409.12186* (2024).

[32] Les Jackson. "Data Transfer Objects". In: *The Complete ASP.NET Core 3 API Tutorial: Hands-On Building, Testing, and Deploying*. Berkeley, CA: Apress, 2020, pp. 191–206. ISBN: 978-1-4842-6255-9. DOI: `10.1007/978-1-4842-6255-9_9`. URL: `https://doi.org/10.1007/978-1-4842-6255-9_9`.

[33] Laurens Jansen. *TomTom - Increasing Developer Productivity using the TiCS Framework*. en-US. URL: `https://www.tiobe.com/knowledge/article/increasing-developer-productivity-using-the-tics-framework/` (visited on 06/14/2025).

[34] Paul Jansen. *Coding Standard Viewer*. URL: `https://csviewer.tiobe.com/#/ruleset/intro?tagid=DIa5r7z2QqCEKHXmHLHfFw&setid=d4441hsNSnyvBQLpRWdAow` (visited on 05/20/2025).

[35] Ziwei Ji et al. "Survey of Hallucination in Natural Language Generation". In: *ACM Computing Surveys* 55.12 (Dec. 2023). arXiv:2202.03629 [cs], pp. 1–38. ISSN: 0360-0300, 1557-7341. DOI: `10.1145/3571730`. URL: `http://arxiv.org/abs/2202.03629` (visited on 05/08/2025).

[36] Juyong Jiang et al. *A Survey on Large Language Models for Code Generation*. arXiv:2406.00515. June 2024. DOI: `10.48550/arXiv.2406.00515`. URL: `http://arxiv.org/abs/2406.00515` (visited on 11/04/2024).

[37] Carlos E. Jimenez et al. *SWE-bench: Can Language Models Resolve Real-World GitHub Issues?* arXiv:2310.06770 [cs]. Nov. 2024. DOI: `10.48550/arXiv.2310.06770`. URL: `http://arxiv.org/abs/2310.06770` (visited on 12/12/2024).

[38] Sathvik Joel, Jie JW Wu, and Fatemeh H. Fard. *A Survey on LLM-based Code Generation for Low-Resource and Domain-Specific Programming Languages*. arXiv:2410.03981 [cs]. Nov. 2024. DOI: `10.48550/arXiv.2410.03981`. URL: `http://arxiv.org/abs/2410.03981` (visited on 12/12/2024).

[39] Nitish Shirish Keskar et al. *CTRL: A Conditional Transformer Language Model for Controllable Generation*. arXiv:1909.05858 [cs]. Sept. 2019. DOI: `10.48550/arXiv.1909.05858`. URL: `http://arxiv.org/abs/1909.05858` (visited on 05/16/2025).

[40] Hyun-Jin Kim, Kwang-Jae Kim, and Doh-Soon Kwak. "A case study on modeling and optimizing photolithography stage of semiconductor fabrication process". en. In: *Quality and Reliability Engineering International* 26.7 (2010). _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/qre.1149, pp. 765–774. ISSN: 1099-1638. DOI: `10.1002/qre.1149`. URL: `https://onlinelibrary.wiley.com/doi/abs/10.1002/qre.1149` (visited on 05/29/2025).

[41] Takeshi Kojima et al. *Large Language Models are Zero-Shot Reasoners*. arXiv:2205.11916 [cs]. Jan. 2023. DOI: `10.48550/arXiv.2205.11916`. URL: `http://arxiv.org/abs/2205.11916` (visited on 12/13/2024).

[42] Sumith Kulal et al. *SPoC: Search-based Pseudocode to Code*. arXiv:1906.04908 [cs]. June 2019. DOI: `10.48550/arXiv.1906.04908`. URL: `http://arxiv.org/abs/1906.04908` (visited on 12/13/2024).

[43] Spraha Kumawat et al. "Sentiment Analysis Using Language Models: A Study". In: *2021 11th International Conference on Cloud Computing, Data Science & Engineering (Confluence)*. Jan. 2021, pp. 984–988. DOI: `10.1109/Confluence51648.2021.9377043`. URL: `https://ieeexplore.ieee.org/abstract/document/9377043` (visited on 10/25/2024).

[44] C. Lambrechts. "Metrics for control models in a model-driven engineering environment". English. PDEng thesis. EngD Thesis. Sept. 2017.

[45] Brian Lester, Rami Al-Rfou, and Noah Constant. *The Power of Scale for Parameter-Efficient Prompt Tuning*. arXiv:2104.08691 [cs]. Sept. 2021. DOI: `10.48550/arXiv.2104.08691`. URL: `http://arxiv.org/abs/2104.08691` (visited on 12/14/2024).

[46] Patrick S. H. Lewis et al. "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks". In: *CoRR* abs/2005.11401 (2020). arXiv: `2005.11401`. URL: `https://arxiv.org/abs/2005.11401`.

[47] Jia Li et al. *AceCoder: Utilizing Existing Code to Enhance Code Generation*. arXiv:2303.17780 [cs]. Sept. 2023. DOI: `10.48550/arXiv.2303.17780`. URL: `http://arxiv.org/abs/2303.17780` (visited on 12/17/2024).

[48] Jia Li et al. *Structured Chain-of-Thought Prompting for Code Generation*. arXiv:2305.06599 [cs]. Sept. 2023. DOI: `10.48550/arXiv.2305.06599`. URL: `http://arxiv.org/abs/2305.06599` (visited on 12/13/2024).

[49] Raymond Li et al. *StarCoder: may the source be with you!* arXiv:2305.06161 [cs]. Dec. 2023. DOI: `10.48550/arXiv.2305.06161`. URL: `http://arxiv.org/abs/2305.06161` (visited on 12/11/2024).

[50] Xiang Lisa Li and Percy Liang. *Prefix-Tuning: Optimizing Continuous Prompts for Generation*. arXiv:2101.00190 [cs]. Jan. 2021. DOI: 10.48550/arXiv.2101.00190. URL: http://arxiv.org/abs/2101.00190 (visited on 12/14/2024).

[51] Xiaonan Li et al. "CodeRetriever: Unimodal and Bimodal Contrastive Learning". In: *CoRR* abs/2201.10866 (2022). arXiv: 2201.10866. URL: https://arxiv.org/abs/2201.10866.

[52] Yujia Li et al. "Competition-Level Code Generation with AlphaCode". In: *Science* 378.6624 (Dec. 2022). arXiv:2203.07814 [cs], pp. 1092–1097. ISSN: 0036-8075, 1095-9203. DOI: 10.1126/science.abq1158. URL: http://arxiv.org/abs/2203.07814 (visited on 12/13/2024).

[53] Chin-Yew Lin. "ROUGE: A Package for Automatic Evaluation of Summaries". In: *Text Summarization Branches Out*. Barcelona, Spain: Association for Computational Linguistics, July 2004, pp. 74–81. URL: https://aclanthology.org/W04-1013 (visited on 12/13/2024).

[54] Fang Liu et al. *Exploring and Evaluating Hallucinations in LLM-Powered Code Generation*. arXiv:2404.00971 [cs]. May 2024. DOI: 10.48550/arXiv.2404.00971. URL: http://arxiv.org/abs/2404.00971 (visited on 05/08/2025).

[55] Jiawei Liu et al. "Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation". en. In: *Advances in Neural Information Processing Systems* 36 (Dec. 2023), pp. 21558–21572. URL: https://proceedings.neurips.cc/paper_files/paper/2023/hash/43e9d647ccd3e4b7b5baab53f0368686-Abstract-Conference.html (visited on 06/13/2025).

[56] Mingwei Liu et al. *Code Copycat Conundrum: Demystifying Repetition in LLM-based Code Generation*. arXiv:2504.12608 [cs]. Apr. 2025. DOI: 10.48550/arXiv.2504.12608. URL: http://arxiv.org/abs/2504.12608 (visited on 05/16/2025).

[57] Tianyang Liu, Canwen Xu, and Julian McAuley. *RepoBench: Benchmarking Repository-Level Code Auto-Completion Systems*. arXiv:2306.03091 [cs]. Oct. 2023. DOI: 10.48550/arXiv.2306.03091. URL: http://arxiv.org/abs/2306.03091 (visited on 12/16/2024).

[58] Shuai Lu et al. *CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation*. arXiv:2102.04664 [cs]. Mar. 2021. DOI: 10.48550/arXiv.2102.04664. URL: http://arxiv.org/abs/2102.04664 (visited on 05/08/2025).

[59] Ziyang Luo et al. *WizardCoder: Empowering Code Large Language Models with Evol-Instruct*. arXiv:2306.08568 [cs]. June 2023. DOI: 10.48550/arXiv.2306.08568. URL: http://arxiv.org/abs/2306.08568 (visited on 12/14/2024).

[60] Varad Maitra, Yutai Su, and Jing Shi. "Virtual metrology in semiconductor manufacturing: Current status and future prospects". In: *Expert Systems with Applications* 249 (Sept. 2024), p. 123559. ISSN: 0957-4174. DOI: 10.1016/j.eswa.2024.123559. URL: https://www.sciencedirect.com/science/article/pii/S095741742400424X (visited on 01/07/2025).

[61] Ggaliwango Marvin et al. "Prompt Engineering in Large Language Models". en. In: *Data Intelligence and Cognitive Informatics*. Ed. by I. Jeena Jacob, Selwyn Piramuthu, and Przemyslaw Falkowski-Gilski. Singapore: Springer Nature, 2024, pp. 387–402. ISBN: 978-981-9979-62-2. DOI: 10.1007/978-981-99-7962-2_30.

[62] Douglas R. Miller. *Markov Processes*. en. Pages: 937-941 Publication Title: Encyclopedia of Operations Research and Management Science. Springer, Boston, MA, 2013. ISBN: 978-1-4419-1153-7. DOI: 10.1007/978-1-4419-1153-7_582. URL: https://link.springer.com/rwe/10.1007/978-1-4419-1153-7_582 (visited on 06/13/2025).

[63] OpenAI et al. *GPT-4 Technical Report*. arXiv:2303.08774 [cs]. Mar. 2024. DOI: 10.48550/arXiv.2303.08774. URL: http://arxiv.org/abs/2303.08774 (visited on 12/11/2024).

[64] Kishore Papineni et al. "Bleu: a Method for Automatic Evaluation of Machine Translation". In: *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*. Ed. by Pierre Isabelle, Eugene Charniak, and Dekang Lin. Philadelphia, Pennsylvania, USA: Association for Computational Linguistics, July 2002, pp. 311–318. DOI: 10.3115/1073083.1073135. URL: https://aclanthology.org/P02-1040 (visited on 12/13/2024).

[65] Dylan Peterson. *Global Semiconductor Sales Increase 18.8% in Q1 2025 Compared to Q1 2024; March 2025 Sales up 1.8% Month-to-Month*. en-US. May 2025. URL: https://www.semiconductors.org/global-semiconductor-sales-increase-18-8-in-q1-2025-compared-to-q1-2024-march-2025-sales-up-1-8-month-to-month/ (visited on 05/29/2025).

[66] Shauli Ravfogel, Yoav Goldberg, and Jacob Goldberger. *Conformal Nucleus Sampling*. arXiv:2305.02633 [cs]. May 2023. DOI: 10.48550/arXiv.2305.02633. URL: http://arxiv.org/abs/2305.02633 (visited on 05/16/2025).

[67] Shuo Ren et al. *CodeBLEU: a Method for Automatic Evaluation of Code Synthesis*. arXiv:2009.10297 [cs]. Sept. 2020. DOI: 10.48550/arXiv.2009.10297. URL: http://arxiv.org/abs/2009.10297 (visited on 12/13/2024).

[68] Stephen Robertson and Hugo Zaragoza. "The Probabilistic Relevance Framework: BM25 and Beyond". In: *Foundations and Trends® in Information Retrieval* 3.4 (2009), pp. 333–389. ISSN: 1554-0669. DOI: 10.1561/1500000019. URL: http://dx.doi.org/10.1561/1500000019.

[69] Pranab Sahoo et al. *A Systematic Survey of Prompt Engineering in Large Language Models: Techniques and Applications*. arXiv:2402.07927 [cs]. Feb. 2024. DOI: 10.48550/arXiv.2402.07927. URL: http://arxiv.org/abs/2402.07927 (visited on 10/14/2024).

[70] Siddharth Samsi et al. "From Words to Watts: Benchmarking the Energy Costs of Large Language Model Inference". In: *2023 IEEE High Performance Extreme Computing Conference (HPEC)*. Boston, MA, USA: IEEE, Sept. 2023, pp. 1–9. ISBN: 9798350308600. DOI: 10.1109/HPEC58863.2023.10363447. URL: https://ieeexplore.ieee.org/document/10363447/ (visited on 06/09/2025).

[71] Douglas Schonholtz. *A Review of Repository Level Prompting for LLMs*. arXiv:2312.10101 [cs]. Dec. 2023. DOI: 10.48550/arXiv.2312.10101. URL: http://arxiv.org/abs/2312.10101 (visited on 12/16/2024).

[72] C. E. Shannon. "A mathematical theory of communication". In: *The Bell System Technical Journal* 27.3 (July 1948). Conference Name: The Bell System Technical Journal, pp. 379–423. ISSN: 0005-8580. DOI: 10.1002/j.1538-7305.1948.tb01338.x. URL: https://ieeexplore.ieee.org/document/6773024 (visited on 10/25/2024).

[73] Mohammed Latif Siddiq, Beatrice Casey, and Joanna C. S. Santos. *FRANC: A Lightweight Framework for High-Quality Code Generation*. arXiv:2307.08220 [cs]. Aug. 2024. DOI: 10.48550/arXiv.2307.08220. URL: http://arxiv.org/abs/2307.08220 (visited on 05/08/2025).

[74] Mohammed Latif Siddiq et al. "An Empirical Study of Code Smells in Transformer-based Code Generation Techniques". In: *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. Limassol, Cyprus: IEEE, Oct. 2022, pp. 71–82. ISBN: 978-1-66549-609-4. DOI: 10.1109/SCAM55253.2022.00014. URL: https://ieeexplore.ieee.org/document/10006873/ (visited on 05/08/2025).

[75] Mukul Singh et al. *CodeFusion: A Pre-trained Diffusion Model for Code Generation*. arXiv:2310.17680 [cs]. Nov. 2023. DOI: 10.48550/arXiv.2310.17680. URL: http://arxiv.org/abs/2310.17680 (visited on 12/13/2024).

[76] Karan Singhal et al. *Large Language Models Encode Clinical Knowledge*. arXiv:2212.13138 [cs]. Dec. 2022. DOI: 10.48550/arXiv.2212.13138. URL: http://arxiv.org/abs/2212.13138 (visited on 05/26/2025).

[77] Aishwarya Suresh. *Model analytics for ASML's data and control modeling languages*. en. Dec. 2018. URL: https://research.tue.nl/en/studentTheses/model-analytics-for-asmls-data-and-control-modeling-languages (visited on 05/09/2025).

[78] Jeniya Tabassum et al. "Code and Named Entity Recognition in StackOverflow". In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL)*. 2020. URL: https://www.aclweb.org/anthology/2020.acl-main.443/.

[79] Florian Tambon et al. *Bugs in Large Language Models Generated Code: An Empirical Study*. arXiv:2403.08937 [cs]. Mar. 2024. DOI: 10.48550/arXiv.2403.08937. URL: http://arxiv.org/abs/2403.08937 (visited on 06/11/2025).

[80] CodeGemma Team et al. *CodeGemma: Open Code Models Based on Gemma*. arXiv:2406.11409. June 2024. DOI: `10.48550/arXiv.2406.11409`. URL: `http://arxiv.org/abs/2406.11409` (visited on 11/04/2024).

[81] *TWINSCAN: 20 years of lithography innovation*. en. Aug. 2021. URL: `https://www.asml.com/en/news/stories/2021/twinscan-20-years-innovation` (visited on 05/29/2025).

[82] Ashish Vaswani et al. *Attention Is All You Need*. arXiv:1706.03762. Aug. 2023. DOI: `10.48550/arXiv.1706.03762`. URL: `http://arxiv.org/abs/1706.03762` (visited on 10/28/2024).

[83] Yizhong Wang et al. *Self-Instruct: Aligning Language Models with Self-Generated Instructions*. arXiv:2212.10560 [cs]. May 2023. DOI: `10.48550/arXiv.2212.10560`. URL: `http://arxiv.org/abs/2212.10560` (visited on 12/14/2024).

[84] Yue Wang et al. *CodeT5+: Open Code Large Language Models for Code Understanding and Generation*. arXiv:2305.07922. May 2023. DOI: `10.48550/arXiv.2305.07922`. URL: `http://arxiv.org/abs/2305.07922` (visited on 11/04/2024).

[85] Jason Wei et al. *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models*. arXiv:2201.11903 [cs]. Jan. 2023. DOI: `10.48550/arXiv.2201.11903`. URL: `http://arxiv.org/abs/2201.11903` (visited on 12/13/2024).

[86] Jason Wei et al. *Finetuned Language Models Are Zero-Shot Learners*. arXiv:2109.01652 [cs]. Feb. 2022. DOI: `10.48550/arXiv.2109.01652`. URL: `http://arxiv.org/abs/2109.01652` (visited on 12/14/2024).

[87] Yuxiang Wei et al. *Magicoder: Empowering Code Generation with OSS-Instruct*. arXiv:2312.02120 [cs]. June 2024. DOI: `10.48550/arXiv.2312.02120`. URL: `http://arxiv.org/abs/2312.02120` (visited on 10/14/2024).

[88] Martin Weyssow et al. *Exploring Parameter-Efficient Fine-Tuning Techniques for Code Generation with Large Language Models*. en. arXiv:2308.10462 [cs]. Jan. 2024. DOI: `10.48550/arXiv.2308.10462`. URL: `http://arxiv.org/abs/2308.10462` (visited on 12/14/2024).

[89] Can Xu et al. *WizardLM: Empowering Large Language Models to Follow Complex Instructions*. arXiv:2304.12244 [cs]. June 2023. DOI: `10.48550/arXiv.2304.12244`. URL: `http://arxiv.org/abs/2304.12244` (visited on 12/14/2024).

[90] Lin Yang et al. *On the Evaluation of Large Language Models in Unit Test Generation*. arXiv:2406.18181 [cs]. Sept. 2024. DOI: `10.48550/arXiv.2406.18181`. URL: `http://arxiv.org/abs/2406.18181` (visited on 06/18/2025).

[91] Junchi Yao et al. *Understanding the Repeat Curse in Large Language Models from a Feature Perspective*. arXiv:2504.14218 [cs]. May 2025. DOI: `10.48550/arXiv.2504.14218`. URL: `http://arxiv.org/abs/2504.14218` (visited on 06/11/2025).

[92] Hao Yu 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*. ICSE '24. New York, NY, USA: Association for Computing Machinery, Feb. 2024, pp. 1–12. ISBN: 9798400702174. DOI: `10.1145/3597503.3623316`. URL: `https://doi.org/10.1145/3597503.3623316` (visited on 12/16/2024).

[93] Zihan Yu et al. *Towards Better Chain-of-Thought Prompting Strategies: A Survey*. arXiv:2310.04959 [cs]. Oct. 2023. DOI: `10.48550/arXiv.2310.04959`. URL: `http://arxiv.org/abs/2310.04959` (visited on 05/29/2025).

[94] Fengji Zhang et al. *RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation*. arXiv:2303.12570 [cs]. Oct. 2023. DOI: `10.48550/arXiv.2303.12570`. URL: `http://arxiv.org/abs/2303.12570` (visited on 12/17/2024).

[95] Haopeng Zhang, Philip S. Yu, and Jiawei Zhang. *A Systematic Survey of Text Summarization: From Statistical Methods to Large Language Models*. arXiv:2406.11289. June 2024. DOI: `10.48550/arXiv.2406.11289`. URL: `http://arxiv.org/abs/2406.11289` (visited on 10/25/2024).

[96] Kechi Zhang et al. *CodeAgent: Enhancing Code Generation with Tool-Integrated Agent Systems for Real-World Repo-level Coding Challenges*. arXiv:2401.07339 [cs]. Aug. 2024. DOI: `10.48550/arXiv.2401.07339`. URL: `http://arxiv.org/abs/2401.07339` (visited on 12/16/2024).

[97] Zhuosheng Zhang et al. *Automatic Chain of Thought Prompting in Large Language Models*. arXiv:2210.03493 [cs]. Oct. 2022. DOI: 10.48550/arXiv.2210.03493. URL: http://arxiv.org/abs/2210.03493 (visited on 12/13/2024).

[98] Wayne Xin Zhao et al. *A Survey of Large Language Models*. arXiv:2303.18223. Sept. 2024. DOI: 10.48550/arXiv.2303.18223. URL: http://arxiv.org/abs/2303.18223 (visited on 10/09/2024).

[99] Qinkai Zheng et al. *CodeGeeX: A Pre-Trained Model for Code Generation with Multilingual Benchmarking on HumanEval-X*. arXiv:2303.17568. July 2024. DOI: 10.48550/arXiv.2303.17568. URL: http://arxiv.org/abs/2303.17568 (visited on 11/03/2024).

[100] Yuqi Zhu et al. *Hot or Cold? Adaptive Temperature Sampling for Code Generation with Large Language Models*. arXiv:2309.02772 [cs]. Dec. 2023. DOI: 10.48550/arXiv.2309.02772. URL: http://arxiv.org/abs/2309.02772 (visited on 05/16/2025).

[101] Terry Yue Zhuo et al. *BigCodeBench: Benchmarking Code Generation with Diverse Function Calls and Complex Instructions*. arXiv:2406.15877 [cs]. Apr. 2025. DOI: 10.48550/arXiv.2406.15877. URL: http://arxiv.org/abs/2406.15877 (visited on 05/16/2025).

A
Prompts used for code generation

This section presents the prompts used to generate the garages, which were created using five different prompting techniques. The detailed prompts for each technique are provided below for reference. The prompts contain certain placeholders that are filled dynamically based on the garage that is being generated.

A.1. Zero-Shot Prompting

Within our codebase we use a type of data service known as garages. Garages can be used to store, retrieve, and update data form the domain model. Since the Durative Actions domain is stateless and has no access to context of data object (its associations to other entities), the garages interfaces contain only value objects and data transfer objects (DTO's). Note that garages don't offer the possibility to delete data. Deletion of data happens by cascaded deletion when the control entity it refers to gets deleted by its Lifecycle. A garage can be of two types, a retrieve garage or a store garage.

We now need to write a new garage. The name of the new garage will be $\{garage\_name\}$. The files that are associated to the new garage are as follows: $\{related\_files\}$.

Complete the $\{garage\_name\}$ class and method declarations as defined in the header file of the garage. Use all the other files as additional information to write the garage. Do not leave any place holders to be filled later. Complete all the methods to the best of your ability. Make sure to thoroughly check all the imports and ensure that you import all the libraries that have been used in the code. Only use methods that are present in the related files. Do not use methods that do not exist. Only return the cpp file associated to this garage without markdown. Do not provide any kind of additional information.

A.2. One-Shot Prompting

Within our codebase we use a type of data service known as garages. Garages can be used to store, retrieve, and update data form the domain model. Since the Durative Actions domain is stateless and has no access to context of data object (its associations to other entities), the garages interfaces contain only value objects and data transfer objects (DTO's). Note that garages don't offer the possibility to delete data. Deletion of data happens by cascaded deletion when the control entity it refers to gets deleted by its Lifecycle. A garage can be of two types, a retrieve garage or a store garage.

We now need to write a new garage. The name of the new garage will be $\{garage\_name\}$. The files that are associated to the new garage are as follows: $\{related\_files\}$.

An example of how a garage can be implemented is given by the following implementation:
$\{example\_garage\}$

Complete the $\{garage\_name\}$ class and method declarations as defined in the header file of the garage. Use all the other files as additional information to write the garage. Do not leave any place holders to be filled later. Complete all the methods to the best of your ability. Make sure to thoroughly check all the imports and ensure that you import all the libraries that have been used in the code. Only use methods that are present in the related files. Do not use methods that do not exist. Only return the cpp file associated to this garage without markdown. Do not provide any kind of additional information.

A.3. Few-Shot Prompting

Within our codebase we use a type of data service known as garages. Garages can be used to store, retrieve, and update data form the domain model. Since the Durative Actions domain is stateless and has no access to context of data object (its associations to other entities), the garages interfaces contain only value objects and data transfer objects (DTO's). Note that garages don't offer the possibility to delete data. Deletion of data happens by cascaded deletion when the control entity it refers to gets deleted by its Lifecycle. A garage can be of two types, a retrieve garage or a store garage.

We now need to write a new garage. The name of the new garage will be $\{garage\_name\}$. The files that are associated to the new garage are as follows: $\{related\_files\}$

An example of how a store garage can be implemented is given by the following implementation:
$\{store\_garage\}$

An example of how a retrieve garage can be implemented is given by the following implementation:
$\{ret\_garage\}$

An example of a store and retrieve garage implemented together is given as follows: $\{store\_ret\_garage\}$

Complete the $\{garage\_name\}$ class and method declarations as defined in the header file of the garage. Use all the other files as additional information to write the garage. Do not leave any place holders to be filled later. Complete all the methods to the best of your ability. Make sure to thoroughly check all the imports and ensure that you import all the libraries that have been used in the code. Only use methods that are present in the related files. Do not use methods that do not exist. Only return the cpp file associated to this garage without markdown. Do not provide any kind of additional information.

A.4. One-Shot Chain-of-Thought Prompting

Within our codebase we use a type of data service known as garages. Garages can be used to store, retrieve, and update data form the domain model. Since the Durative Actions domain is stateless and has no access to context of data object (its associations to other entities), the garages interfaces contain only value objects and data transfer objects (DTO's). Note that garages don't offer the possibility to delete data. Deletion of data happens by cascaded deletion when the control entity it refers to gets deleted by its Lifecycle. A garage can be of two types, a retrieve garage or a store garage.

A garage can be implemented by following multiple steps. An example of how a garage can be implemented is given by the following implementation: $\{example\_garage\}$

The steps followed to construct this garage are as follow: **The actual steps used to construct the garage have been redacted due to confidentiality reason.**

We now need to write a new garage. The name of the new garage will be $\{garage\_name\}$. The files that are associated to the new garage are as follows: $\{related\_files\}$.

Complete the $\{garage\_name\}$ class and method declarations as defined in the header file of the garage by following the steps as described above. Use all the other files as additional information to write the garage. Do not leave any place holders to be filled later. Complete all the methods to the best of your ability. Make sure to thoroughly check all the imports and ensure that you import all the libraries that have been used in the code. Only use methods that are present in the related files. Do not use methods that do not exist. Only return the cpp file associated to this garage without markdown. Do not provide any kind of additional information.

A.5. Few-Shot Chain-of-Thought Prompting

Within our codebase we use a type of data service known as garages. Garages can be used to store, retrieve, and update data form the domain model. Since the Durative Actions domain is stateless and has no access to context of data object (its associations to other entities), the garages interfaces contain only value objects and data transfer objects (DTO's). Note that garages don't offer the possibility to delete data. Deletion of data happens by cascaded deletion when the control entity it refers to gets deleted by its Lifecycle. A garage can be of two types, a retrieve garage or a store garage.

A garage can be implemented by following multiple steps. An example of how a store garage can be implemented is given by the following implementation: $\{store\_garage\}$

The steps followed to construct this garage are as follow: **The actual steps used to construct the garage have been redacted due to confidentiality reason.**

An example of how a retrieve garage can be implemented is given by the following implementation: $\{ret\_garage\}$

The steps followed to construct this garage are as follow: **The actual steps used to construct the garage have been redacted due to confidentiality reason.**

An example of how a store and retrieve garage can be implemented is given by the following implementation: $\{store\_ret\_garage\}$

The steps followed to construct this garage are as follow: **The actual steps used to construct the garage have been redacted due to confidentiality reason.**

We now need to write a new garage. The name of the new garage will be $\{garage\_name\}$. The files that are associated to the new garage are as follows: $\{related\_files\}$.

Complete the $\{garage\_name\}$ class and method declarations as defined in the header file of the garage by following the steps as described above. Use all the other files as additional information to write the garage. Do not leave any place holders to be filled later. Complete all the methods to the best of your ability. Make sure to thoroughly check all the imports and ensure that you import all the libraries that have been used in the code. Only use methods that are present in the related files. Do not use methods that do not exist. Only return the cpp file associated to this garage without markdown. Do not provide any kind of additional information.