UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
UNIDADE ACADÊMICA DE SISTEMAS E COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

DÉBORA LÊDA DE LUCENA SOUZA

EVALUATING LARGE AND SMALL LANGUAGE MODELS FOR PROGRAMMING PROBLEM SOLVING

CAMPINA GRANDE - PB

2025

Universidade Federal de Campina Grande

Centro de Engenharia Elétrica e Informática

Coordenação de Pós-Graduação em Ciência da Computação

Evaluating Large and Small Language Models for Programming Problem Solving

Débora Lêda de Lucena Souza

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação

Linha de Pesquisa: Engenharia de Software

Rohit Gheyi

(Orientador)

Campina Grande, Paraíba, Brasil

Débora Lêda de Lucena Souza

Evaluating Large and Small Language Models for Programming Problem Solving

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande, pertencente à linha de pesquisa de Engenharia de Software, área de concentração Ciência da Computação, como requisito para a obtenção do Título de Mestre em Ciência da Computação.

Aprovado(a) em: 24/02/2025

S729e Souza, Débora Lêda de Lucena.
Evaluating large and small language models for programming problem solving / Débora Lêda de Lucena Souza. – Campina Grande, 2025.
90 f. : il. color.

Dissertação (Mestrado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2025.
"Orientação: Prof. Dr. Rohit Gheyi".
Referências.

1. Software Engineering. 2. Large Language Models. 3. Small Language Models. 4. Code Generation. 5. Code Generation Evaluation. I. Gheyi, Rohit. II. Título.

UFCG/BC CDU 004.41(043.3)

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECÁRIA SEVERINA SUELI DA SILVA OLIVEIRA CRB-15/225

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
POS-GRADUACAO EM CIENCIA DA COMPUTACAO
Rua Aprígio Veloso, 882, Edifício Telmo Silva de Araújo, Bloco CG1, - Bairro Universitário, Campina Grande/PB, CEP 58429-900
Telefone: 2101-1122 - (83) 2101-1123 - (83) 2101-1124
Site: http://computacao.ufcg.edu.br - E-mail: secpg@computacao.ufcg.edu.br

FOLHA DE ASSINATURA PARA TESES E DISSERTAÇÕES

DÉBORA LÊDA DE LUCENA SOUZA

EVALUATING LARGE AND SMALL LANGUAGE MODELS FOR PROGRAMMING PROBLEM SOLVING

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação como pré-requisito para obtenção do título de Mestre em Ciência da Computação.

Aprovada em: 24/02/2025

Prof. Dr. ROHIT GHEYI, Orientador, UFCG

Prof. Dr. MÁRCIO DE MEDEIROS RIBEIRO, Examinador Interno, UFAL

Prof. Dr. GUSTAVO ARAÚJO SOARES, Examinador Externo, MICROSOFT

Resumo

A transformação de linguagem natural em código está evoluindo rapidamente, impulsionada por avanços em Grandes e Pequenos Modelos de Linguagem (LLMs e SLMs). Embora demonstrem grande potencial na geração de código, a eficácia desses modelos em cenários reais de programação ainda é incerta, especialmente considerando diferentes tipos de problemas e níveis de dificuldade. Este estudo avalia a acurácia de Grandes Modelos de Linguagem (GPT-4, LLAMA 3, CLAUDE 3 SONNET e GEMINI PRO 1.0) em 100 problemas do LeetCode e BeeCrowd, além de investigar o desempenho de Pequenos Modelos de Linguagem (LLAMA 3.2 3B, GEMMA 2 9B, PHI-4 14Be DEEPSEEK-R1 14B) em 280 problemas do Codeforces. Os resultados mostram que, no grupo de LLMs, o GPT-4 liderou com 78 soluções corretas, evidenciando maior facilidade em problemas de nível mais baixo. Já entre os SLMs, o PHI-4 14B destaca-se ao resolver 63% dos problemas, superando significativamente os outros modelos, que apresentaram taxas inferiores a 23%. Esses achados indicam o potencial dos LLMs e SLMs como assistentes de codificação, mas também ressaltam a variação significativa nas taxas de sucesso conforme a complexidade dos problemas. Portanto, apesar de auxiliarem de forma relevante, não devem ser adotados como soluções autônomas. No caso dos SLMs, embora o PHI-4 14B apresente resultados promissores, ainda há limitações.

Abstract

The transformation of natural language into code is evolving rapidly, driven by advances in Large and Small Language Models (LLMs and SLMs). Although it demonstrates great potential in code generation, the effectiveness of these models in real programming scenarios is still uncertain, especially considering different types of problem and levels of difficulty. This study evaluates the accuracy of Large Language Models (GPT-4, LLAMA 3, CLAUDE 3 SONNET and GEMINI PRO 1.0) on 100 LeetCode and BeeCrowd problems, in addition to investigating the performance of Small Language Models (LLAMA 3.2 3B, GEMMA 2 9B, PHI-4 14Band DEEPSEEK-R1 14B) on 280 Codeforces problems. The results show that, in the group of LLMs, GPT-4 led with 78 correct solutions, showing greater ease in lower-level problems. Among SLMs, PHI-4 14B stands out by solving 63% of problems, significantly outperforming other models, which apply rates lower than 23%. These results indicate the potential of LLMs and SLMs as settlement residents, but also highlight the significant variation in success rates depending on the complexity of the problems. Therefore, despite helping significantly, they should not be adopted as independent solutions. In the case of SLMs, although PHI-4 14B presents promising results, there are still limitations.

Agradecimentos

Agradeço à minha família, pelo amor incondicional, paciência e por sempre estarem ao meu lado, me incentivando e me proporcionando o apoio necessário em todos os momentos. Aos meus irmãos, que sempre acreditaram em meu potencial e me ensinaram a perseverar, muito obrigada.

Ao meu esposo, pela compreensão, paciência e por estar ao meu lado nos momentos mais difíceis. Sua presença constante, apoio emocional e incentivo foram fundamentais para o meu crescimento pessoal e acadêmico.

Aos meus amigos, que com palavras de apoio, incentivo e amizade, tornaram essa jornada mais leve e alegre. Obrigada por sempre estarem presentes, mesmo à distância, e por me motivarem a continuar.

Agradeço também ao meu orientador, Rohit Gheyi, por todo o apoio, orientações e ensinamentos que me proporcionaram ao longo do desenvolvimento deste trabalho. Sua paciência e dedicação foram essenciais para a evolução desta pesquisa.

Por fim, agradeço às agências de fomento, CAPES e CNPq, pelo apoio recebido ao longo do meu percurso acadêmico. As ações dessas instituições de fomento são fundamentais para a construção de uma base sólida e para o desenvolvimento da pesquisa no Brasil.

A todos que, de alguma forma, contribuíram para a realização deste trabalho, meu sincero agradecimento.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

According to Uptech [101], developing software typically costs between \$50,000 and \$250,000, depending on complexity and development hours, with multi-platform applications often reaching the higher end. As software development becomes more accessible, the global developer population has grown to \$19.6 million in 2024, according to JetBrains report [47]. However, hiring and retaining skilled developers remains costly, with rates ranging from \$24 to \$150 per hour depending on experience, location, and employment model [4]. Driven by soaring demand for top-tier talent, these expenses contribute to the broader trend of rising IT investment, with global spending projected to reach \$5.26 trillion in 2024—a 7.5% increase from the previous year, according to Gartner [32].

Given the high costs of software development and retaining skilled professionals, there is a growing need for cost-reduction strategies—one of the most promising being automatic code generation, which leverages large code bases to generate code from developer-specified requirements, enabling faster and more scalable development. Automatic code generation has evolved rapidly, from early methods using natural language inputs [61, 113] to powerful models like CodeBERT [28] and OpenAI's CodeX [14], which leverage large-scale pretraining to support multilingual code tasks and drive tools like Copilot.

Building upon these advanced models, the last two years have witnessed the emergence of several sophisticated Large Language Models (LLMs) tailored for code generation tasks. Notable examples include GPT-4 [2], GEMINI PRO 1.0 [98], LLAMA 3 [34], and CLAUDE 3 SONNET [3]. Despite their strong performance, these large models require massive parameter counts, leading to high computational demands during training and deployment. This results in elevated operational costs, energy use, and carbon emissions, raising environmental concerns [88, 94, 95]. Moreover, LLMs pose security risks such as data leakage and vulnerability to adversarial attacks, threatening privacy and sensitive information [23].

To address economic, environmental, and security concerns, researchers are increasingly focusing on Small Language Models (SLMs), which use fewer parameters to reduce computational demands, costs, carbon impact and are easily customizable. Their open-source nature also enables local deployment and better privacy. Despite their size, recent SLMs like PHI-4 14B [1], LLAMA 3.2 3B [66], GEMMA 2 9B [99], and DEEPSEEK-R1 14B [38] show potential for code generation.

1.1 Problem

The rapid advancement of LLMs and SLMs has positioned them as powerful tools for automated code generation, enabling natural language-to-code translation that reduces costs, increases productivity, and expands access to software development. However, a critical challenge lies in reliably assessing the correctness of the code these models generate. Despite their widespread use, benchmarks like HumanEval [14], MBPP [7], and APPS [39] have notable limitations [26, 48]: HumanEval offers few test cases and overly simplified prompts; MBPP focuses on basic Python tasks; and APPS, while more complex, evaluates only code correctness, ignoring key metrics like execution time and memory usage. Moreover, dataset-based evaluations may produce false positives, accept inefficient solutions, and struggle with problems that allow multiple correct outputs [59].

Given these considerations, it is crucial to evaluate both LLMs and SLMs on a broader and more challenging set of tasks, particularly within real-world competitive programming environments. Platforms like the Codeforces Judge [17], which demand efficient algorithms, strict execution constraints, and scalable solutions, offer a valuable setting for such assessments. Evaluating models in these dynamic, high-stakes contexts allows for a more comprehensive understanding of their true capabilities and limitations—measuring not only correctness but also runtime and memory usage. This approach provides a more practical and realistic evaluation, helping bridge the gap between academic benchmarks and real-world software engineering needs.

1.2 Motivating Example

Evaluating the correctness of code generated by language models is challenging due to the complexity and diversity of programming tasks, which are influenced by factors like problem difficulty, required concepts (e.g., arrays, matrices, sorting), necessary optimizations, and the clarity of the prompt. Additionally, a language model can produce correct but inefficient solutions and struggle with problems that have multiple valid outputs.

An illustrative example of the complexity involved in automatic code generation can be found in problem 414-B (Elo rating 1400) from Codeforces, Listing 1.1. In this problem, the goal is to count how many fixed-length sequences can be formed using numbers from 1 to n, such that each element divides the next— a constraint that requires not only generating all possible combinations but also efficiently checking divisibility between consecutive elements. The solution demands a dynamic programming approach that carefully considers the relationship between divisors and multiples, effectively illustrating the kind of algorithmic reasoning language models must master to solve non-trivial programming tasks.

This solution was generated by the Phi-4 model and, although it produces correct results, it fails to meet the time constraints imposed by the platform. Specifically, it successfully passes the first four test cases but exceeds the time limit on the fifth, resulting in a Time Limit Exceeded (TLE) error.

1.3 Solution

To address this gap, our study presents an evaluation of LLMs—including GPT-4, GEMINI PRO 1.0, LLAMA 3, and CLAUDE 3 SONNET—and SLMs, such as PHI-4 14B, LLAMA 3.2 3B, GEMMA 2 9B, and DEEPSEEK-R1 14B. We assess their performance across multiple dimensions: correctness, runtime and memory usage, consistency, and cost-effectiveness. The evaluation is conducted using a diverse set of competitive programming problems sourced from globally recognized platforms—LeetCode [56], BeeCrowd [9], and Codeforces [17]—which are widely used by millions of programmers and serve as industry benchmarks for algorithmic problem-solving. These problems span a broad range of topics and difficulty levels, from basic data structures to advanced algorithmic challenges, providing a

```python
def count_good_sequences(n, k):
    MOD = 1000000007
    dp = [[0 for _ in range(k + 1)] for _ in range(n + 1)]

    for i in range(1, n + 1):
        dp[i][1] = 1

    for length in range(2, k + 1):
        for end in range(1, n + 1):
            total = 0
            multiple = end
            while multiple <= n:
                total += dp[multiple][length - 1]
                if total >= MOD:
                    total -= MOD
                multiple += end
            dp[end][length] = total

    result = sum(dp[i][k] for i in range(1, n + 1)) % MOD
    return result

if __name__ == "__main__":
    import sys
    input = sys.stdin.read
    data = list(map(int, input().split()))
    n, k = data[0], data[1]
    print(count_good_sequences(n, k))
```

Listing 1.1: Python implementation for the *Mashmokh and ACM* problem (Codeforces 414-B)

comprehensive view of each model's coding capabilities. Ultimately, our analysis offers a holistic assessment of the effectiveness of LLMs and SLMs in automated code generation, shedding light on their strengths, limitations, and implications for the future of AI-assisted programming.

By adopting this approach, we aim to derive valuable insights into the performance of

LLMs and SLMs in competitive programming. Our analysis focuses on several key aspects, including overall accuracy rates—measuring how often a model generates a fully correct solution that passes all test cases—and accuracy across different difficulty levels, which helps us understand how these models perform on easy, medium, and hard problems. We also examine common error patterns in the generated code, such as logical mistakes or syntax issues, and evaluate the number of attempts required by LLMs to produce a correct solution, highlighting whether success tends to occur on the first try or after multiple retries. For SLMs, we assess consistency by determining whether correct solutions are produced reliably across multiple runs or merely by chance. Finally, we consider the financial cost of running SLMs in competitive programming environments to evaluate their cost-effectiveness.

1.4 Evaluation

Our research methodology consists of two complementary experiments designed to evaluate the performance of LLMs and SLMs in solving competitive programming problems. The first experiment focuses on LLMs and involves a manual evaluation using a curated set of 100 problems, 50 from LeetCode and 50 from BeeCrowd. Solutions generated by the models are submitted directly to the respective platforms, ensuring correctness is assessed under real-world competitive conditions. To account for variability in model outputs, each problem is submitted up to three times. The second experiment targets SLMs and uses an automated evaluation framework applied to a larger dataset of 280 problems from Codeforces. This approach streamlines the testing process by extracting problem requirements and submitting them systematically, with each problem evaluated three times per model—totaling 840 submissions per model. Together, these experiments enable a thorough assessment of correctness, consistency, runtime, and computational efficiency. This methodology provides an evaluation of the coding capabilities of LLMs and SLMs, ensuring that their performance is assessed in a fair and reproducible manner.

1.5 Conclusions

Our findings reveal meaningful differences in the performance of LLMs and SLMs on code generation tasks, particularly when considering problem difficulty and platform characteristics. For instance, while LLMs like GPT-4 performed well overall, with high accuracy on structured problems—achieving over 90% success on LeetCode—they showed a notable drop on more open-ended tasks, with average accuracy falling to around 40% on BeeCrowd. This contrast underscores how model performance is influenced not only by problem complexity, but also by how tasks are and presented.

In the automated evaluation of SLMs on Codeforces problems, models like PHI-4 14B showed promising capabilities, reaching a pass@3 rate above 60%, while others performed significantly below that threshold. This stark contrast highlights the considerable performance gap between different sizes of small models, emphasizing that while some SLMs demonstrate promising capabilities, others still struggle to generalize effectively across diverse coding challenges

Overall, the study highlights that while LLMs and SLMs can serve as effective coding assistants, their utility depends heavily on task type, clarity of the problem description, and the underlying reasoning required. Selected models show strong potential, but ensuring code correctness, reliability, and practical value in real-world activities demands careful validation and testing. Moving forward, continued evaluation in realistic, high-variance programming environments will be essential to fully understand and improve their real-world applicability.

1.6 Summary of contributions

The main contributions of this work are:

- A comparative analysis of the correctness and reliability of LLMs, including GPT-4, GEMINI PRO 1.0, LLAMA 3, and CLAUDE 3 SONNET, in solving programming problems.

- A comparative analysis of the correctness and reliability of SLMs, such as PHI-4 14B, LLAMA 3.2 3B, GEMMA 2 9B and DEEPSEEK-R1 14B, in solving programming problems.

1.7 Organization

This work is organized as follows: Chapter 2 provides the essential background information. Chapters 3 and 4 present the methodology and results of the evaluation for Large and Small Language Models, respectively. Chapter 5 offers a review of related work, and Chapter 6 concludes with a summary of the findings.

Chapter 2

Background

This chapter provides a thorough background to facilitate a deeper understanding of the concepts underpinning our study. It is structured into three key sections: an introduction to Language Models (Section 2.1), an exploration of Code Generation Techniques (Section 2.2), and a discussion on the Evaluation of Language Models Code Generated (Section 2.3).

2.1 Introduction to Language Models

In this section, we provide an overview of the development of language models, tracing their evolution from early models to the advent of Large Language Models (LLMs). We will also delve into the key trends, innovations, and factors that have shaped the development of LLMs, alongside the emergence and significance of smaller language models as an alternative to their larger counterparts.

2.1.1 The Evolution of Language Models

Machines cannot inherently comprehend or generate human language in a meaningful way without sophisticated artificial intelligence (AI) models. Over the years, significant advancements in natural language processing (NLP) have driven the development of Large Language Models (LLMs), enabling machines to interpret, generate, and interact using human language [100]. However, achieving human-like proficiency in programming tasks remains a challenge, requiring continuous refinement of model architectures, training methodologies, and evaluation frameworks.

Language modeling (LM) is a key approach in enhancing the language intelligence of machines. In general, LM focuses on estimating the generative probability of word sequences, allowing for the prediction of future (or missing) tokens. This field has garnered significant attention in research, evolving through four major developmental stages [119]:

- **Statistical Language Models (StatLM):** Emerging in the 1990s, StatLMs [31,83,96] use statistical methods to predict the next word in a sentence based on preceding words. These models rely on the *Markov assumption*, considering only a limited number of previous words for prediction. When a fixed number of words is used as context, they are known as *n-gram models* (e.g., bigram for two words, trigram for three). However, StatLMs struggle with large datasets, as the number of possible word combinations grows exponentially, making accurate predictions more difficult. To address this issue, smoothing techniques like *backoff estimation* [53] and *Good-Turing estimation* [30] are used to adjust probabilities for unseen word combinations, improving prediction accuracy.

- **Neural Language Models (NLM):** NLMs [12, 54, 70] use neural networks to predict the likelihood of word sequences. Neural networks like multi-layer perceptrons (MLP) and recurrent neural networks (RNNs) are employed for this task. One significant advancement introduced by these models was the idea of representing words as vectors [12], which helps to capture the meaning of words based on the context in which they appear. By applying this technique of learning features, a general neural network approach was developed to create a unified and efficient solution for various natural language processing (NLP) tasks [20]. Additionally, the word2vec [69, 71] model was proposed to build a simple neural network that learns these word representations, and this method proved to be highly effective across various NLP tasks. These studies marked the beginning of using language models for representation learning, beyond just modeling word sequences, having a significant impact on the field of NLP.

- **Pre-trained Language Models (PLM):** A key development in this area was ELMo [78], which aimed to improve word representations by first training a network to understand the context of words and then adjusting it for specific tasks. Instead of using fixed word meanings, ELMo used a bidirectional LSTM (biLSTM) network. Following this, BERT [24] introduced a more advanced approach using the Transformer model [102], which allowed the network to learn from large amounts of text data through a process called self-attention. BERT improved the understanding of word context and provided better word representations that helped boost the performance of many NLP tasks. This approach of "pre-training and fine-tuning" has since been widely adopted, leading to the creation of other models like GPT-2 [81] and BART [57], which either introduced new network designs or enhanced the pre-training process. Fine-tuning is often necessary to adapt these pre-trained models to specific tasks.

- **Large language models (LLM):** Larger language models (LLMs) tend to perform better on various tasks as they grow in size and data, following a scaling law [52]. Researchers have tested this by training increasingly bigger models, like GPT-3 (175B parameters) and PaLM (540B parameters), which show emergent abilities [108] compared to smaller models like BERT (330M) and GPT-2 (1.5B). These larger models can handle complex tasks, such as few-shot learning, which smaller ones struggle with. A key example is ChatGPT, which adapts GPT models for conversation, demonstrating impressive interaction skills. Since its release, research on LLMs has surged on the computer science community.

2.1.2 Large versus Small Language Models

LLMs are trained on massive text corpora with tens of billions (or more) of parameters, such as GPT-3 [51], GPT-4 [2], and LLAMA 3 [34]. The goal of LLMs is to enable machines to understand human commands and adhere to human values. The substantial increase in model size, dataset volume, and computational prowess has resulted in significant enhancements across various tasks and unveiled remarkable capabilities. The number of parameters for LLMs typically exceeds a hundred billion, and the training data is usually in the range of a few hundred GB to a few TB. The largest version of the GPT-4 model has 1.76 trillion parameters and uses hundreds of terabytes of text data for training. This enormous scale allows these models to capture a broad spectrum of knowledge and perform sophisticated tasks, including natural language understanding, code generation, summarization, and more, often outperforming smaller models in accuracy and versatility.

Despite the progression of Language Models towards Large Language Models (LLMs), many tasks can still be effectively handled by smaller versions of these models. While the computational infrastructure required to run an LLM is not always accessible to everyone, and not all tasks necessitate the use of such large models, companies have responded by releasing more compact versions of their LLMs. These smaller models, such as Phi-4:14B [68], Llama3.2:3B [67], and Gemma2:9B [33], offer a balanced trade-off between performance and efficiency, making them accessible for a broader range of users and tasks. These models demonstrate that, with innovative approaches and optimized data handling, smaller models can still achieve remarkable results while being more feasible to deploy in resource-constrained environments.

The Phi-4 model was developed in response to recent advancements in Large Language Models (LLMs), which have shown that improving data quality can lead to performance gains that rival or even surpass those achieved by simply scaling computational resources, model size, or dataset volume. Phi-4, a 14-billion parameter model, pushes the boundaries of small language models by introducing innovative synthetic data generation techniques, specifically designed for reasoning-intensive tasks. By optimizing its training curriculum, fine-tuning data mixtures, and employing novel post-training strategies, Phi-4 significantly enhances the performance of smaller models [1].

A key innovation behind Phi-4 is its use of synthetic data, which forms the majority of its training set. This data is generated through a range of sophisticated techniques, such as multi-agent prompting, self-revision workflows, and instruction reversal. These approaches facilitate the creation of datasets that promote advanced reasoning and problem-solving capabilities, overcoming some of the limitations found in traditional unsupervised data sources. As a result, Phi-4 demonstrates strong reasoning abilities, making it highly effective in tackling tasks that demand complex cognitive functions [1].

Thanks to these groundbreaking methods, Phi-4's performance on reasoning-based tasks often rivals, or even exceeds, that of much larger models. For instance, in many widely recognized reasoning benchmarks, Phi-4 competes directly with Llama-3.1-405B, showcasing the significant strides that smaller models have made in achieving comparable or superior results despite their reduced parameter count [1].

2.2 Code Generation Techniques

Code generation, also known as program synthesis, refers to the automated generation of software code based on user intent. This technique enhances developer productivity by reducing manual coding and accelerating the software development lifecycle. Early research in code generation focused primarily on deductive and inductive program synthesis, which creates code based on specifications and/or input-output pairs. With the advent of deep learning techniques, language-based code generation, which describes user intent in natural language, has gained significant attention [64].

2.2.1 Deductive Code Generation

Deductive synthesis relies on logical reasoning to derive a program from a specification. In this process, the specification is typically a formal description of the desired program behavior (such as expected inputs and outputs), and the system deduces a program that satisfies these conditions. Deductive synthesis is often associated with formal logic techniques and formal proofs, where the machine attempts to derive a correct program based on a formal description of requirements and properties [35, 37].

Example of operation: If the problem specification indicates that given an integer, the program should return its double, deductive synthesis would attempt to construct a program that satisfies this condition based on known logical and mathematical rules.

The need for detailed specifications aids in minimizing logical errors. It has wide-ranging applications in areas such as robotics [29] and software engineering [41]. For instance, STRIPS [29] is an automated planner that tackles robot-related challenges, while PROW [103] generates LISP code from specifications in predicate calculus by employing a two-step process involving theorem proving and code generation.

2.2.2 Inductive Code Generation

Inductive program synthesis, also known as programming by example (PBE), creates programs directly from specific input-output pairs [64]. This method is simpler and more accessible than deductive synthesis, and it has been widely explored. It allows users with little to no programming experience to guide computers using examples.

Example of operation: If the problem specification provides examples such as the number 2 generating the output 4 and the number 3 generating the output 6, inductive synthesis would attempt to deduce that the desired program is a function that doubles the input values.

For example, FlashFill [36], one of the most well-known real-world applications of program synthesis, generates programs for spreadsheet software like Excel based on just a few input-output examples. Similar techniques [106] are also employed to generate programs for relational databases, particularly for tasks like schema refactoring.

2.2.3 Natural Language-based Code Generation

Natural language-based code generation has become a significant area of research, utilizing deep learning techniques to convert natural language descriptions into functional code. These approaches are typically divided into three main categories: sequence-based models, tree-based models, and pre-trained models [64].

Sequence-based models treat code generation as a machine translation problem, where the task is to convert a natural language description into code. For example, Ling et al. [61] used a neural network with a structured attention mechanism to process semi-structured inputs for code generation.

Tree-based models take into account the structured nature of code by parsing it into tree-like structures, such as Abstract Syntax Trees (ASTs). Yin et al. [114] trained an LSTM to generate a sequence of actions that build the AST, while Rabinovich et al. [80] directly generate the tree structure of the source code.

Pre-trained models [28,105] have emerged as another effective approach, where models are pre-trained on large datasets and later fine-tuned for specific tasks like code generation. These models have led to significant improvements in performance. Additionally, some studies have explored "retrieval-augmented generatio" where the model retrieves relevant code from external sources such as Stack Overflow or API documentation to enhance the generation process. For instance, Xu et al. [111] incorporated external knowledge bases for improved model performance, while Parvez et al. [77] introduced similar code snippets alongside the input to train models in incorporating reusable code.

These approaches represent a broad range of strategies that are advancing the field of natural language-based code generation, each contributing to the overall improvement of code generation capabilities.

2.2.4 Language Models for Code Generation

Language Models (LMs) for code generation involve using LMs to produce source code based on natural language descriptions, a task commonly referred to as natural-language-to-code. These descriptions usually consist of programming problem statements (or docstrings) and may also include additional programming context, such as function signatures and assertions [48].

Transformer-based LMs have transformed numerous fields, with a significant impact on code generation. Their development follows a structured process, beginning with the curation and synthesis of code data, followed by a multi-stage training pipeline that includes pre-training, fine-tuning (instruction tuning), and reinforcement learning with various feedback mechanisms. Additionally, advanced prompt engineering techniques enhance their effectiveness. Recent innovations have introduced repository-level and retrieval-augmented code generation, along with the emergence of autonomous coding agents [48].

2.3 Evaluation of Language Models Code Generated

To assess the performance and advantages of LLMs, strategies and benchmarks have been introduced to facilitate empirical evaluation and analysis.

2.3.1 Evaluation Strategies

Developing effective and reliable automatic evaluation metrics for generated content has been a persistent challenge in natural language processing (NLP) [15, 60, 76]. Initially, most approaches relied on token-matching-based metrics, such as Exact Match, BLEU [76], ROUGE [60], and METEOR [8], which are widely used in NLP text generation, to evaluate the quality of code generation [48].

Although these metrics provide a fast and cost-effective way to evaluate generated code, they often fail to fully capture its syntactic and functional correctness, as well as its semantic properties. To address this limitation, CodeBLEU [82] was introduced, extending the traditional BLEU metric [76] by integrating syntactic analysis through abstract syntax trees (AST) and semantic understanding via data-flow graphs (DFG). Despite these advancements, the metric still falls short in addressing execution errors and discrepancies in the generated code's execution results. To overcome these challenges, execution-based metrics have gained traction in code generation evaluation, including pass@k [14], n@k [58], test case average [40], and pass@t [74]. Among these, pass@k has emerged as a key evaluation metric, measuring the likelihood that at least one of k generated code samples successfully passes all unit tests. An unbiased estimator for pass@k, introduced by [14], is defined as:

$$\texttt{pass@k} = 1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \tag{2.1}$$

Let n represent the total number of sampled candidate code solutions, k the number of randomly selected code solutions from these candidates for each programming problem, where $n \geq k$, and c the number of correct solutions within the k selected samples.

However, execution-based methods rely significantly on the quality of unit tests and are restricted to evaluating executable code [116]. As a result, when unit tests are not available, token-matching-based metrics are frequently used as an alternative for evaluation. In cases where a ground truth label is missing, unsupervised metrics like perplexity (PPL) [46] can be applied. Perplexity measures an LLM's uncertainty in predicting new content, offering an indirect assessment of the model's generalization ability and the quality of the generated code.

2.3.2 Code Generation Benchmarks

To rigorously evaluate the effectiveness of Language Models (LMs) in code generation, the research community has developed a diverse set of high-quality benchmarks in recent years.

These benchmarks, including various iterations of the HumanEval dataset and other novel benchmarks, aim to assess a wider range of code generation capabilities in large language models. By incorporating increasingly complex problems and augmenting test case scales, these benchmarks provide a comprehensive framework for measuring the performance of LLMs in producing functional, accurate, and contextually appropriate code [48].

HumanEval [14] consists of 164 manually created Python programming problems, each containing a function signature, docstring, body, and multiple unit tests. HumanEval+ [63] expands on the original HumanEval benchmark by scaling up the number of test cases by a factor of 80. This increased scale enables HumanEval+ to detect a larger amount of incorrect code generated by LLMs that may have previously gone unnoticed.

MBPP [7] consists of around 974 Python programming problems, crowdsourced and aimed at entry-level programmers. Each problem includes an English task description, a code solution, and three automated test cases. MBPP+ [63] builds upon MBPP by removing poorly structured problems and correcting those with faulty implementations. Additionally, MBPP+ significantly increases the test scale, expanding it by 35 times for enhanced test coverage.

CoderEval [115] is a practical code generation benchmark featuring 230 Python and 230 Java programming problems. It serves to assess model performance in generating functional code that goes beyond simple standalone functions. ClassEval [26], on the other hand, is a manually designed benchmark with 100 classes and 412 methods aimed at evaluating LLMs in the context of class-level code generation. The tasks in ClassEval are particularly complex, requiring the generation of long code and detailed docstrings, making it a valuable tool for evaluating the ability of LLMs to produce intricate and sophisticated code.

For competition code, there is APPS benchmark [40] which is composed of 10K Python problems, spanning three levels of difficulty: introductory, interview, and competition. Each problem is described in English, accompanied by its corresponding ground truth Python solutions, and test cases defined by their inputs and outputs or function names when provided. CodeContests [58], on the other hand, is a competitive programming dataset that includes samples from various platforms, such as Aizu, AtCoder, CodeChef, Codeforces, and HackerEarth. This dataset encompasses programming problems along with test cases presented as paired inputs and outputs, including both correct and incorrect human solutions across multiple programming languages.

Additionally, LiveCodeBench [45] is a comprehensive and contamination-free benchmark for evaluating a wide range of code-related capabilities of LLMs, including code generation, self-repair, code execution, and test output prediction. The benchmark continuously gathers new coding problems from reputable contest platforms, such as LeetCode, AtCoder, and CodeForces. The latest release of the dataset contains 713 problems, which were published between May 2023 and September 2024.

Chapter 3

Manual Evaluation of Large Language Models (LLMs)

In this chapter, we outline the methodology, results, and discussions of our study on Large Language Models, which involved evaluating 100 problems from LeetCode and BeeCrowd. Section 3.1 describes the methodology employed, while Section 3.2 defines the objectives of our evaluation, including the research questions and performance metrics used. Section 3.3 presents the main findings, followed by Section 3.4, which offers an in-depth analysis and interpretation of these results. Finally, Section 3.5 addresses the potential threats to the validity of our study.

3.1 Methodology

This section outlines the methodology employed to develop the work and accomplish the set objectives. It is organized into several subsections: Platforms Selection (Section 3.1.1), Problems Selection (Section 3.1.2), Large Language Models Selection (Section 3.1.3), Prompt Design (Section 3.1.4), Experiment Setup (Section 3.1.5), Pilot Study (Section 3.1.6), and Experiment Phases (Section 3.1.7).

3.1.1 Platforms Selection

The LeetCode [56] and BeeCrowd [9] platforms were selected because they are an well-known repository of programming problems, widely used to enhance coding skills, participate in contests, and prepare for job interviews. These platforms offer a range of problems that simulate real-world challenges companies may face in their daily operations, with sufficiently difficult tasks to thoroughly test the capabilities of Language Models.

3.1.2 Problems Selection

To conduct the Manual Evaluation of Large Language Models analysis, 100 programming problems were manually selected to evaluate whether the GPT-4, GEMINI PRO 1.0, LLAMA 3, and CLAUDE 3 SONNET models provide correct responses. Of these, 50 problems were sourced from LeetCode and 50 from BeeCrowd, both popular programming platforms. The problems span across Easy, Intermediate, and Hard difficulty levels, and were chosen arbitrarily and manually submitted to the LLMs prompts.

The LeetCode platform categorizes its problems into difficulty levels: Easy, Intermediate, and Hard. On the other hand, the BeeCrowd platform categorizes its problems by difficulty level, from 1 to 10. To facilitate comparison with the LeetCode problems, the questions were redistributed into three semantic levels, namely:

- **Easy**: questions from level 1 to 4;
- **Intermediate**: questions from level 5 to 7;
- **Difficult**: questions from level 8 to 10.

Given that the selected problems span Easy, Intermediate, and Hard difficulty levels, we aimed for a thorough evaluation by arbitrarily selecting 50 problems from each platform. These were distributed as follows: 20 from the Easy level, 15 from the Intermediate level, and 15 from the Hard level.

Once each platform sets the level of each problem, a problem with the level "hard" from LeetCode may not have the same level of difficulty compared with a problem from BeeCrowd. Therefore, the problems cover various computer science domains including arrays, data structures, algorithms, and graphs, among others. It is important to note that on

BeeCrowd, the difficulty level is estimated using a variation of the ELO Rating System [72], which uses the number of times the problem has been "defeated" (i.e., how many users have solved the problem) to determine its difficulty. In this system, problems that are solved by many users with few attempts are classified as low difficulty, while problems with more attempts but fewer solutions are classified as high difficulty. The difficulty of the problems is adjusted weekly, so their total score may vary [11]. The questions were selected arbitrarily in January 2023 because preliminary studies involving GPT-3.5 were conducted throughout that year, as detailed in Section 3.1.6.

3.1.3 Large Language Models Selection

The LLMs selected for evaluation in this study were chosen based on their performance and impact in the AI field. We used tools such as LLM Arena [6], a crowdsourced open platform for LLM evaluations, to guide our selection based on their rankings. Additionally, we aimed to include one LLM from different companies and architecture to ensure representation across different developers.

The selected models are GPT-4 [2], LLAMA 3 [34], CLAUDE 3 SONNET [3], and GEMINI PRO 1.0 Pro [98]. As shown in Table 3.1, each one of them was developed by leading AI companies. GPT-4, developed by OpenAI, has 1.76 trillion parameters and a context window of 8k, released in March 2023 with a cutoff in September 2021 [2]. Meta's LLAMA 3 model has 70 billion parameters, a large context window of 128k, and was released in April 2024, with data available up until December 2023 [34]. Anthropic's CLAUDE 3 SONNET also features 70 billion parameters but has an even larger context window of 200k, released in March 2024, with a cutoff date in August 2023 [3]. Google's GEMINI PRO 1.0, with 1.56 trillion parameters, supports an impressive 1 million context window and was released in December 2023 [19] [98].

3.1.4 Prompt Design

To submit the selected problems to LLMs, the complete text of each problem was extracted from the respective platforms and incorporated into the LLMs' prompts. The prompt was carefully designed to provide clear instructions, ensuring that the models understood the task

Table 3.1: Overview of Large Language Models: Context Window, Parameters, and Release Information.

Model Name	Company	Number of Parameters	Context Window	Cutoff Date	Release Date
GPT-4	OpenAI	1.76T	8k	September 2021	March 2023
LLAMA 3	Meta	70B	128k	December 2023	April 2024
CLAUDE 3 SONNET	Anthropic	70B	200k	August 2023	March 2024
GEMINI PRO 1.0	Google	1.56T	32k	-	December 2023

and generated relevant solutions. The following instruction was used:

> Make a program that solves the following problem. The problem statement includes a description, input requirements, output specifications, and examples to aid in understanding the problem.
> Problem Statement: {*Problem Statement*}.

This structured approach ensured consistency across different problems while enabling LLMs to accurately interpret and generate code solutions. Based on the provided input, the language model produces a response code, which is then submitted to the problem platform. The platform serves as an oracle, verifying the correctness of the LM generated solution.

3.1.5 Experiment Setup

The Manual Evaluation of Large Language Models experiment utilized BeeCrowd and LeetCode both as repositories of programming problems and as oracles to validate solutions generated by the models. The models evaluated in this study were GPT-4 (default ChatGPT version), GEMINI PRO 1.0, LLAMA 3, and CLAUDE 3 SONNET. A set of questions was chosen arbitrarily in January 2023, with GEMINI PRO 1.0 being evaluated in February 2024, followed by GPT-4, LLAMA 3, and CLAUDE 3 SONNET in April 2024.

To assess each problem, the full text—spanning description, examples, and constraints was manually copied from the platform and pasted into each model's prompt. In response, the model produced a solution, which was then submitted back to the platform for evaluation. Each solution received an "accepted" or "rejected" verdict based on the platform's test cases. This submission-and-verification workflow, is summarized in Figure 3.1. Although the example in Listing 1.1 does not include an explanatory image, it is common for problem statements to contain images. However, this feature was not included for evaluation alongside the question in any LLM, once the images only clarifies the problem statement instead of add new information.

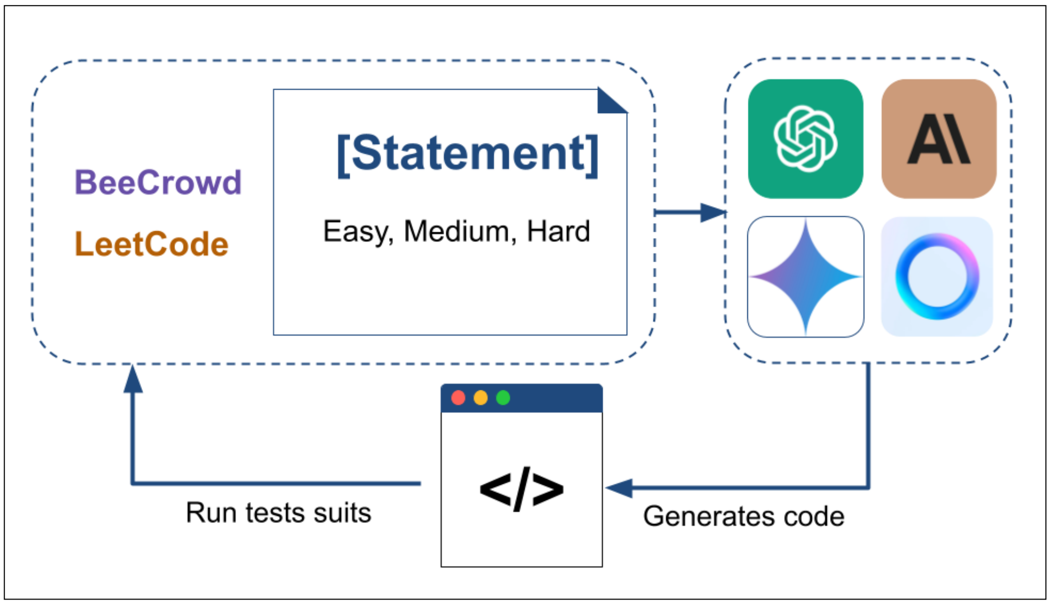


Figure 3.1: Setup of manual evaluation of Large Language Models (LLMs).

Given the probabilistic nature of the models used in this study, they can produce varied responses to the same query. As a result, each LLM was given up to three attempts per problem. This means the identical text was resubmitted for evaluation a maximum of three times, or until a correct answer was obtained. The decision to allow three submissions was purely experimental, with the same prompt being forwarded to the model each time. If the response remained incorrect after the third submission, the problem was classified as unsolved. Each problem was presented to the model in a unique prompt, which was only reused for additional attempts if needed. No specific programming language was requested in the prompts; however, most solutions were generated in Python.

3.1.6 Pilot Study

In this stage, a qualitative pilot study was initially conducted using only the GPT-3.5 LLM, followed later by the GEMINI PRO 1.0. The datasets used in this study were sourced from the LeetCode and BeeCrowd platforms, and the methodology followed is described in Section 3.1.5. The results obtained from this pilot study were incorporated into the analysis, as the studies were published across 2023 and 2024 [90] [92]. These experiments were carried out entirely manually to ensure the validity of the manual study methodology.

3.1.7 Experiment Phases

Following the pilot study, the manual evaluation phase was broadened to incorporate additional LLMs, including GPT-4, LLAMA 3, and CLAUDE 3 SONNET. The same set of problems from LeetCode and BeeCrowd used in the pilot was retained to maintain consistency, enabling direct comparisons of performance and reasoning across different versions and generations of language models.

This expanded phase aimed to enhance the generality and robustness of our approach by examining a wider range of models. By comparing outputs from models trained on different data and employing varied architectures, the study provided deeper insights into their behavior, strengths, and potential pitfalls. These findings were especially valuable for validating the methodology on a broader scale, highlighting model-specific advantages and limitations while informing future refinements of our experimental design.

3.2 Definition

In this section, we define the objective of our study using the **GQM** (Goal, Question, Metric) method [13]. Our goal is to **evaluate** the effectiveness of Large Language Models (LLMs) such as GPT-4, GEMINI PRO 1.0, LLAMA 3, and CLAUDE 3 SONNET in solving programming problems sourced from platforms like LeetCode and BeeCrowd, **with the purpose** of raising implications of its use in the daily routine of programmers, regarding the correctness of the generated answers, **from the perspective of** researchers, **in the context of** automatic code generation.

Therefore, this study aims to provide insights into the practical implications of using LLMs in programmers' daily workflows, with a focus on solution correctness. To achieve this, we formulate and investigate three key research questions (RQs).

RQ$_1$ **To what extent LLMs as GPT-4, LLAMA 3, CLAUDE 3 SONNET, GEMINI PRO 1.0 can answer programming assignments?**

To answer this question, the correct and incorrect responses provided by the platforms LeetCode and BeeCrowd will be counted.

RQ$_2$ **What types of errors are most common in the responses generated by LLMs?**
To address this question, we will examine and categorize the most frequent failures encountered in model outputs, including syntactic issues, logical missteps, and failures to satisfy problem constraints.

RQ$_3$ **How does the performance of LLMs vary across different programming topics?**
This question aims to assess the effectiveness of LLMs across a diverse set of programming topics, including graph algorithms, dynamic programming, sorting, and string manipulation. By analyzing performance variations, we can identify strengths and weaknesses in the models' problem-solving capabilities across different algorithmic domains.

3.3 Research Questions Results

This section presents the results of the manual evaluation conducted on 100 programming problems sourced from BeeCrowd and LeetCode. The analysis primarily focuses on assessing the performance of different LLMs, identifying patterns in their problem-solving capabilities, and highlighting their strengths and limitations. While pilot studies involving GPT-3.5 provided valuable qualitative insights, they were excluded from the quantitative analysis to focus solely on the performance of more recently released models.

3.3.1 RQ$_1$: To what extent LLMs as GPT-4, LLAMA 3, CLAUDE 3 SONNET, GEMINI PRO 1.0 can answer programming assignments?

To answer this research question, the number of correct answers provided by each programming platform was counted, as shown in Table 3.2. Specifically, GPT-4 successfully solved 100% of the programming problems from LeetCode, but only 56% in BeeCrowd. In comparison, LLAMA 3 and CLAUDE 3 SONNET showed similar results, solving 92% of the problems on LeetCode but only 38% on BeeCrowd, indicating a significant variation in performance depending on the platform. GEMINI PRO 1.0 performed similarly on LeetCode, correctly answering 90% of the problems, but struggled on BeeCrowd, achieving only 34% accuracy. These results emphasize that while all models perform well on LeetCode, their effectiveness on BeeCrowd varies significantly.

Table 3.2: Number of problems correctly answered by GPT-4, LLAMA 3, CLAUDE 3 SONNET and GEMINI PRO 1.0.

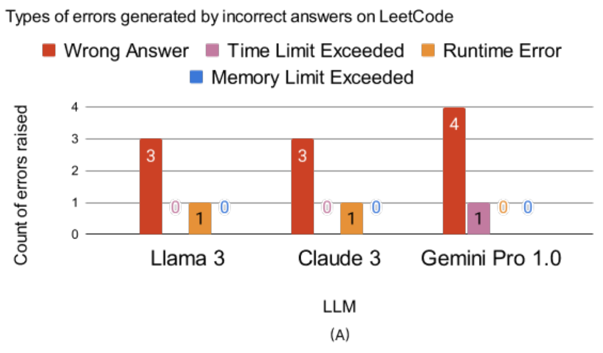
Platform	**GPT-4**	**LLAMA 3**	**CLAUDE 3 SONNET**	**GEMINI PRO 1.0**
LeetCode	50/50	46/50	46/50	45/50
BeeCrowd	28/50	19/50	17/50	17/50
Total	78/100 (78%)	65/100 (65%)	63/100 (63%)	62/100 (55%)

3.3.2 RQ$_2$ What types of errors are most common in the responses generated by the models?

The LeetCode and BeeCrowd platforms provide an output with the associated error for each problem that is not solved. Figure 3.2 groups together all the errors recorded by each platform, the main errors are Time Limit Exceeded and Wrong Answer. Time Limit Exceeded is an error thrown when the submitted solution takes longer than the allowed time to execute all evaluation tests [11]. On the other hand, Wrong Answer is thrown when the solution does not produce the expected result for 100% of the test cases.

Additionally, there is also the Runtime Error, thrown in smaller quantities, which concerns defining a vector or array with less capacity than required for the problem, or attempting to access an invalid memory location. Memory Limit Exceeded is generated when the code attempts to allocate more memory than the maximum allowed for the problem. This can occur because a very large vector or data structure is being used [11].

On the LeetCode platform, the model GPT-4 exhibited flawless performance, resulting in its absence from Figure 3.2 (A). In contrast, the models LLAMA 3, CLAUDE 3 SONNET, and GEMINI PRO 1.0 primarily encountered the 'Wrong Answer' error. As illustrated in Figure 3.2 (B), on the BeeCrowd platform the models GEMINI PRO 1.0, CLAUDE 3 SONNET,



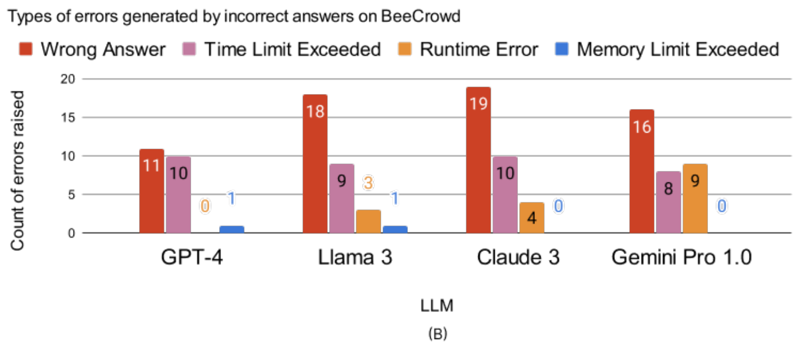


Figure 3.2: Types of errors generated by incorrect answers on (A) - LeetCode platform vs (B) - BeeCrowd platform.

and LLAMA 3 frequently encounter the 'Wrong Answer' error, indicating that their solutions often fail the test cases. Conversely, GPT-3.5 and GPT-4 predominantly experience the 'Time Limit Exceeded' error, suggesting that their incorrect solutions do not complete within the allotted time for executing test cases.

The range of types of errors exhibited by GPT-4 enforces a diversity of responses offered by the model. In contrast, most of the incorrect answers provided by GEMINI PRO 1.0, CLAUDE 3 SONNET and LLAMA 3 share the same error "Wrong answer," indicating that the model consistently makes the same type of mistake.

3.3.3 RQ$_3$ How does the performance of LLMs vary across different programming topics?

The LeetCode and BeeCrowd platforms provide information about the topics covered in each question, allowing for the assessment of the topics addressed. Tables 3.3 and 3.4 groups together the main topics addressed in the questions discussed in this study. It is notable that

Table 3.3: Number of correct answers per topic on the LeetCode platform.

Topics	**GPT-4**	**Llama 3**	**Claude 3 Sonnet**	**Gemini Pro 1.0**
Array	17	16	17	16
String	10	9	8	8
Hash Table	6	6	6	6
Linked List	6	5	5	5
Math	6	5	5	5
Tree	3	3	3	3
Backtracking	1	1	1	1
Stack	1	1	1	1
Total	50 (100%)	46 (92%)	46 (92%)	45 (90%)

the LeetCode topics covered are widely recognized in the programming world, including Array, String, and Math. Some problems also make use of more complex concepts, such as dynamic programming and recursion. Nevertheless, the models achieved an accuracy rate of 95

The topics covered on BeeCrowd platform are also common in the programming world, such as Data Structures and Libraries, Math, and Paradigms. The most covered topics are Ad-hoc and Beginner, categories created by the platform to target problems that do not fit into other categories and are basic enough for programming beginners [10], respectively.

When examining the topics covered, it is evident that the BeeCrowd platform questions do not address significantly different or more complex topics compared to the LeetCode platform questions. This suggests that the lack of appropriate responses from the models is not due to the topic of the questions, but rather to the formulation of the questions, as BeeCrowd is a competition-oriented platform and many of its questions are intentionally written in a complex manner to make the challenge more offensive.

3.4 Discussion

Building upon the findings outlined in the preceding section, this section delves into various aspects related to the performance of the models. We dig into deeper to identify the diffi-

Table 3.4: Number of correct answers per topic on the BeeCrowd platform.

Topics	**GPT-4**	**Llama 3**	**Claude 3 Sonnet**	**Gemini Pro 1.0**
Ad-hoc	5	2	1	1
Beginner	8	7	7	7
Data Structures and Libraries	5	3	2	3
Graph	-	-	-	-
Math	5	3	4	4
Paradigms	2	1	-	-
Strings	3	3	3	2
Total	28 (56%)	19 (38%)	17(34%)	17 (34%)

culty levels of the questions that are answered correctly and incorrectly. Furthermore, we performed a metamorphic test, analyzed the number of attempts required to correctly solve a question, and explore the implications of utilizing models like GPT-4, Gemini Pro 1.0, Llama 3 and Claude 3 Sonnet for code generation. We also analyze the performance disparity of these models across the platforms LeetCode and BeeCrowd. Additionally, we acknowledge the limitations of our study and outline potential directions for future research.

3.4.1 The Number of Attempts Required to Correctly Solve a Question

Given that each problem was allowed a maximum of three attempts, we can analyze the accuracy rate in relation to the number of submissions required for a correct solution. This evaluation provides insights into the models' consistency, highlighting how often they generate correct answers on the first attempt, versus requiring multiple tries to succeed.

LeetCode Questions

Figure 3.3 (A) compares the performance of GPT-4, Llama 3, Claude 3 Sonnet, and Gemini Pro 1.0 in solving problems from LeetCode. The x-axis of the chart represents the number of attempts made to correctly answer the questions, while the y-axis shows the number of questions solved correctly. Llama 3 and Claude 3 Sonnet have identical performance per attempt, so their lines overlap in the graph.

For the LeetCode platform, most problems were solved on the first attempt, with only about 20% of the problems requiring a second or third attempt. Despite this, a significant portion of the remaining problems were eventually solved, with GEMINI PRO 1.0 achieving the highest success rate at 8/13 (61.54%). This highlights the importance of submitting problems multiple times to LLMs, as their probabilistic nature allows alternative answers suggested by the model to may be correct.

BeeCrowd Questions

Figure 3.3 (B) compares the performance of GPT-4, LLAMA 3, CLAUDE 3 SONNET, and GEMINI PRO 1.0 in solving problems from BeeCrowd. The x-axis represents the number of attempts made to correctly answer questions, while the y-axis shows the number of questions answered correctly.

In the BeeCrowd results, the models solved less than 50% of the problems on the first attempt, and their performance on subsequent attempts was also low. In this context, the model with the highest success rate on remaining attempts was LLAMA 3, achieving a rate of 8 out of 39 (20.51%). On the other hand, GPT-4 had the lowest success rate, solving only 3 out of 39 (7.69%). This analysis concludes that, despite the probabilistic nature of the models, they do not always provide correct answers, regardless of the number of attempts.

3.4.2 Difficulty Levels of the Problems Answered

As shown in Figures 3.4 (A) and (B), GPT-4 exhibited superior problem-solving performance across both evaluated platforms. It successfully solved all 50 problems from LeetCode and correctly answered 28 out of 50 problems from BeeCrowd, each within a maximum of three attempts. An analysis of Figure 3.4 (B) for the BeeCrowd platform reveals a clear trend across all the LLMs examined. As illustrated, problems categorized as Easy were solved with the highest accuracy, followed by Medium and Hard problems. This pattern contrasts with the results from LeetCode, where problems across all difficulty levels were answered correctly.

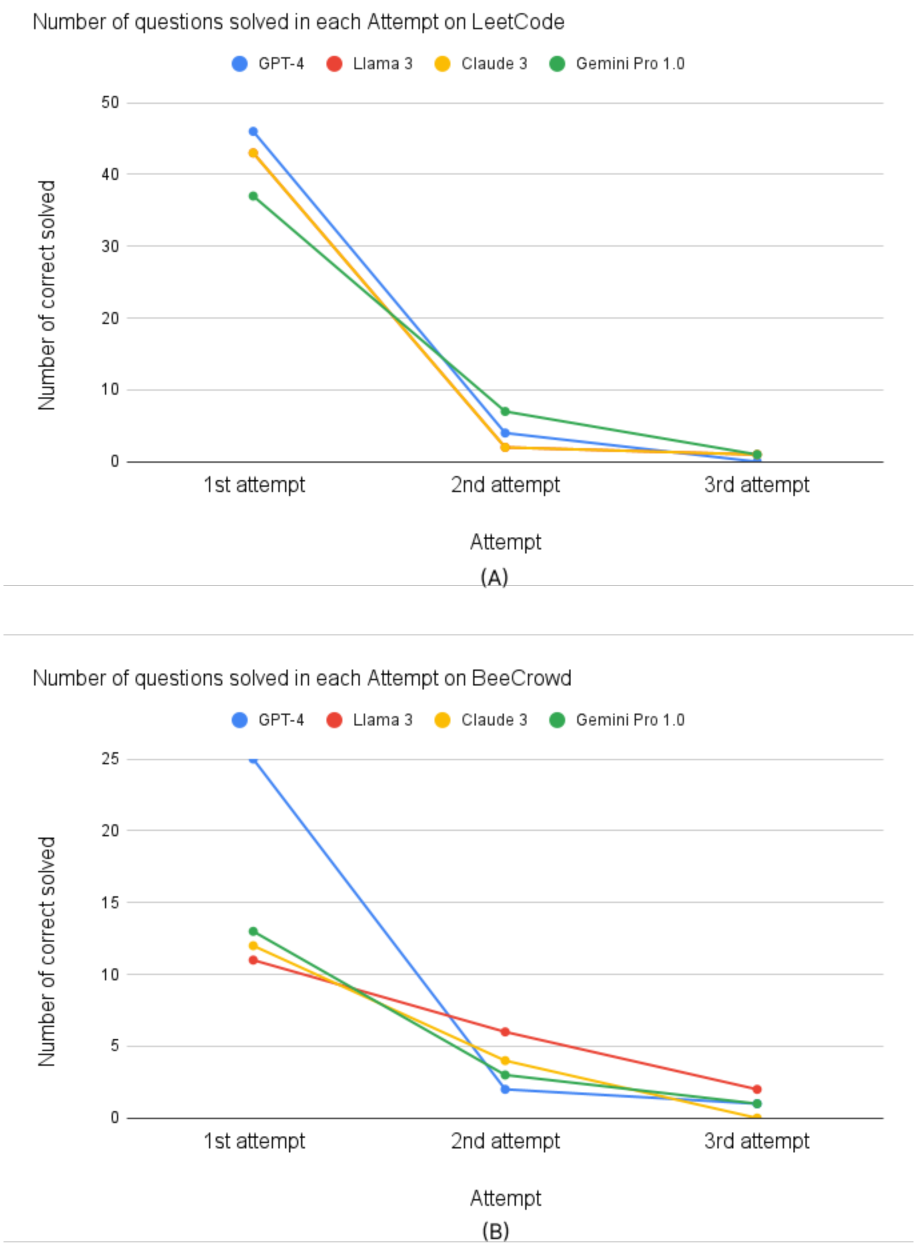
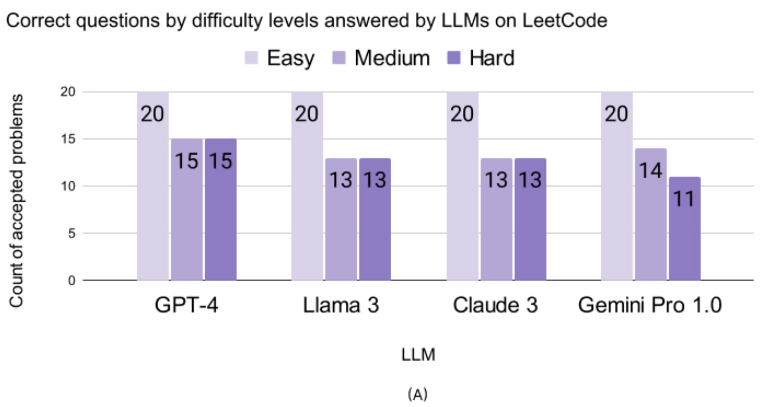


Figure 3.3: Number of correct answers per number of attempts on (A) - LeetCode platform vs (B) - BeeCrowd platform.

3.4.3 Evaluating Data Leakage on GPT-4

One potential threat to validity when using foundation models is data contamination [86]. To mitigate this risk, we apply metamorphic testing [5,16] to assess the robustness and reliability of the models. The GPT-4, GPT-3.5, GEMINI PRO 1.0, CLAUDE 3 SONNET, and LLAMA 3 demonstrate greater efficiency, particularly in solving problems of medium and low complex-



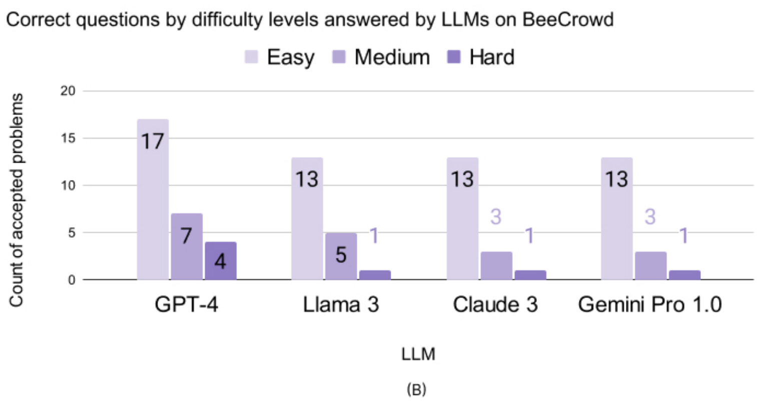


Figure 3.4: Number of correct questions by difficulty levels on (A) - LeetCode platform and (B) - BeeCrowd platform.

ity. However, they encounter significant challenges when addressing problems of medium and high difficulty. The models' lower-than-expected performance on the BeeCrowd platform can likely be attributed to their training data. It is possible that these models included data from LeetCode during their training, a globally popular platform, while excluding data from BeeCrowd, which is less widely recognized internationally.

To investigate further, a Metamorphic Test was conducted using all 50 problems from BeeCrowd problems to assess whether GPT-4 would maintain its response trend. In this study, a Metamorphic Test entails modifying the problem descriptions by:

- Substituting certain words with synonyms,
- Replacing verbs,
- Renaming personal nouns or variables,
- Making minor changes without altering the core meaning of the problem.

For example, original phrases like *"Calculate the sum of X and Y"* were modified to *"Determine the total of A and B"*. This ensures that the fundamental problem remains unchanged, while its linguistic structure is altered. In this example, the words replaced are shown in Table 3.5. After modifying some words, the reformulated requirement document is submitted to the LLM for processing. The response generated by GPT-4 is then submitted to the BeeCrowd platform, and its correctness is recorded for subsequent analysis.

Table 3.5: Example of words changed on metamorphic test.

Word Before	Word After
Calculate	Determine
X	A
Y	B
Sum	Total

The Metamorphic Test was applied to all 50 BeeCrowd problems using the GPT-4 model. As illustrated in Figure 3.5, the number of correctly solved problems before and after the metamorphic test is very similar, with a difference of only three problems, and

Table 3.6: Transition Matrix with status changes before and after the Metamorphic Test.

From/To	Right	Wrong
Right	26	2
Wrong	5	17

there is no significant variation in the count by submission attempts. This aligns with the expected behavior of a probabilistic model, as discussed in Section 3.4.1, where resubmitting a problem may lead to different outcomes.

Table 3.6 presents the Transition Matrix, comparing the status of problems before and after the Metamorphic Test, highlighting the number of problems that changed or maintained their status. In this analysis, 7 problems had their status altered after the Metamorphic Test, which can be attributed to the probabilistic behavior of the model.

In conclusion, since there were no significant variations in the responses provided by the model after the Metamorphic Test, we can infer, within this context and for the set of problems used, that there was no data leakage or contamination between the test data and the training data.

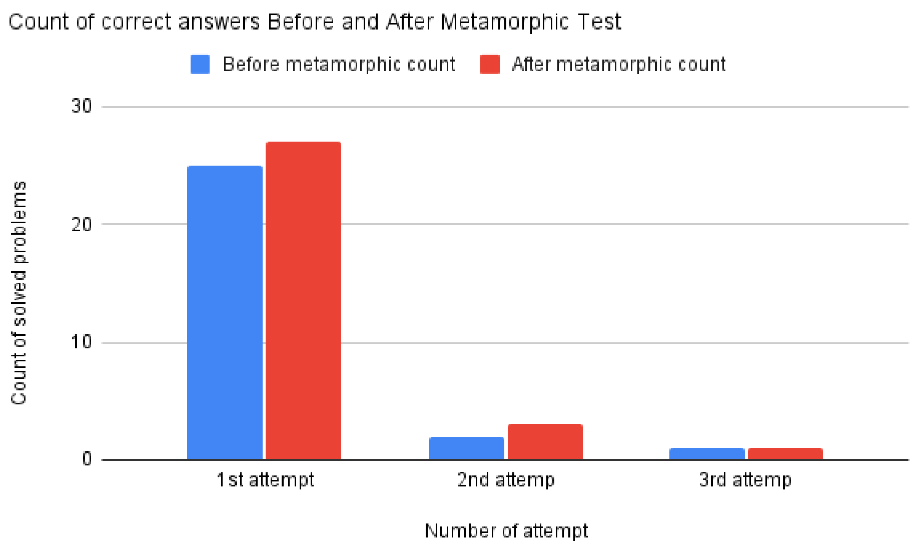


Figure 3.5: Count of correct answers before and after Metamorphic Test.

Table 3.7: Problems released after the cutoff date submitted to GPT-4.

Contest	Solved count
Codeforces 940 Div2 (Easy, Medium)	6/9
Atcoder 351 (Easy - Hard)	2/7
Atcoder 353 (Easy - Hard)	2/7
OPI PB 2024 (Easy, Medium)	5/6

Analysing GPT-4 on Contest After the Cutoff Date

To further investigate potential data leakage, problems created after the LLM's cutoff date were submitted to GPT-4. Since this data could not have been included in GPT-4's training set, it serves as a critical test. Problems were sourced from contests such as Codeforces, Atcoder, and the Paraíba Informatics Olympiad 2024, covering a range of difficulties from easy to hard. However, it is important to note that the difficulty level is assigned by the respective platform. This means that an "Easy" problem on Atcoder may have a significantly higher difficulty level compared to an “Easy” problem on LeetCode.

As demonstrated in Table 3.7, the GPT-4 model successfully solved 15 out of the 29 newly submitted problems, reflecting a moderate level of performance with fresh problems, similar to its performance with BeeCrowd problems. This suggests that its performance is not influenced by data leakage.

3.4.4 Requirement Document Analysis on Not Solved Problems by GPT-4

A subset of the problems that GPT-4 did not solve was analyzed by an Olympic student to understand the underlying reasons for their failure. It was determined that the Requirement Documents for 5 from 13 unsolved problems were ambiguous or incomplete. Figure 3.6 illustrates a problem involving the management of an airport's flight queue. Initially, the description ambiguously suggests that flights from the West should be queued first, followed by those from the North, South, and East. However, the examples clarify that the correct approach is to queue one flight from each region in sequence. To clarify this for GPT-4, a sentence was added to the Requirement Document: “[...] To create the queue, take one flight

Problem Flying Control

Description

[...] In order to organize the flow of airplanes from an airport [...], the planes that come from the West side have a higher priority of being placed in the takeoff or landing queue [...]. Airplanes coming from the North and South side must be inserted in row 1 at a time. And finally, the airplanes coming from the East side.

Input

The entrance consists of an integer P, representing the cardinal point of the plane ($-4 \leq P \leq -1$), where (-4 is East, -3 is North, -2 is South, and -1 is West). Then the planes are entered [...].

Output

The exit consists of a line containing the aircraft lined up in the order.

Examples

Input Sample 1	Input Sample 2
-4 A1 A26 A38 A23	-4 A12 A33
-1 A80 A40	-3 A8 A33
Output Sample 1	Output Sample 2
A80 A1 A40 A26 A38 A23	A8 A12 A33 A33

Note

For example, the boys can divide the watermelon into two parts of 2 and 6 kilos respectively (another variant — two parts of 4 and 4 kilos).

Figure 3.6: Example of a problem with ambiguous Requirement Document.

from each direction in the following order: west, north, south, and east [...].” With such adjustments, 4 out of the 5 problematic cases were resolved.

Enhancing the clarity of the Requirement Document to reduce ambiguity does not guarantee that a problem will be correctly solved. However, a key insight is that unsolved problems, particularly those classified as Easy or Medium, may often stem from ambiguous or incomplete Requirement Documents. Such cases warrant further investigation to improve problem comprehension and solution accuracy.

3.5 Threats to Validity

The study on the performance of LLMs like GPT-4, GEMINI PRO 1.0, LLAMA 3 and CLAUDE 3 SONNET in generating code from natural language descriptions, while illuminating, faces some threats to its validity. These threats can be categorized into internal and external validity threats, alongside construct and conclusion validity concerns:

3.5.1 Internal Validity

Problem Selection Bias: The programming problems selected from platforms like LeetCode and BeeCrowd might not cover all possible types of programming challenges, or may favor certain problem-solving paradigms. This could skew the evaluation towards models that perform better on these specific types of problem.

Formatting Loss: The problems were submitted in a textual form, meaning the text was copied from the platform and pasted into the prompt of the evaluated LLM. Therefore, formatting loss may occur. To ensure consistency, the author preserved the original structure and format of the problem as presented on the platform, making an effort not to add or remove any line break.

Evaluation Criteria: Regarding the assessments of submissions, we assume that the platform's answers are correct. This approach is based on the widespread use of these platforms by various developers. Any failure or error in the correction of questions would likely be identified and reported by the community, allowing for timely correction of these issues.

Comparison Between Test Data and Training Data: The lower performance on BeeCrowd may be due to data extraction restrictions on the site, raising doubts about model's access for training. In contrast, LeetCode has no such restrictions, supported by Gemini Pro's behavior citing LeetCode as references in its answers. To address this threat to validity, the Session 3.4.3 was held to identify data leakage in GPT-4.

Complexity of Problem Requirement Documents: A notable observation is that many unsolved problems originate from the BeeCrowd platform, which exhibits a higher complexity level in its problem statements compared to LeetCode platform. To prevent this, we addressed many different topics of problems, as well as different levels from different platforms.

3.5.2 External Validity

Generalization to Real-world Programming: The programming problems used in the study might not accurately reflect the complexity and diversity of real-world programming tasks. Thus, the models' performance in this controlled setting may not directly translate to effectiveness in practical coding scenarios.

Evolution of Models: These AI models are rapidly evolving, with newer versions being released frequently. The findings may quickly become outdated, limiting the generalization of the study's conclusions over time.

3.5.3 Construct Validity

Evaluation of Correctness: How correctness is defined and measured in solving programming problems can significantly affect the outcomes. How our metric for success does not comprehensively capture the quality of the code generated in terms of efficiency, readability, or adherence to best practices, it may not accurately reflect the models' true capabilities.

Difficulty Level Misclassification Threat Another threat arises from the misclassification of difficulty levels between different platforms. In the study, problems from various platforms such as LeetCode, Atcoder, Codeforces, and BeeCrowd were included, each with its own system of classifying difficulty. An "easy" problem in one platform might correspond to a "medium" or even "hard" problem on another platform, as difficulty classification is subjective and context-dependent. If the difficulty levels are not appropriately accounted for or standardized, the model's performance could be misinterpreted, thereby affecting the construct validity. In order to mitigate this, the study carefully selected a wide range of problems from different platforms, including Codeforces, Atcoder, and BeeCrowd, covering multiple difficulty levels (easy, medium, and hard). This broad selection aimed to provide a balanced representation of problems from each platform, ensuring that the model's performance could be evaluated across various levels of complexity.

Chapter 4

Automated Evaluation of Small Language Models: Findings and Discussion

In this chapter, we outline the methodology, results, and discussions of our study on SLMs, conducted using a dataset of 280 problems from Codeforces. Section 4.1 describes the methodology employed in the study. Section 4.2 presents the objectives of our evaluation, including the research questions and the performance metrics used to assess the models. Section 4.3 summarizes the key results, while Section 4.4 offers a detailed analysis and discussion of the findings.

4.1 Methodology

This section outlines the methodology employed to develop the work and accomplish the set objectives. It is organized into several subsections: Platforms Selection (Section 4.1.1), Problems Selection (Section 4.1.2), Small Language Models Selection (Section 4.1.3), Prompt Design (Section 4.1.4), Experiment Setup (Section 4.1.5), Pilot Study (Section 4.1.6), and Experiment Phases (Section 4.1.7).

4.1.1 Platforms Selection

The Codeforces platform was chosen for this study due to its reputation as a prominent repository of programming problems, commonly used to sharpen coding skills, compete in contests, and prepare for job interviews. It offers a diverse set of problems that mirror real-world challenges encountered by companies, with tasks that are challenging enough to rigorously assess the capabilities of Language Models. Recent research [27] has also leveraged Codeforces problems to evaluate language models, as competitive programming is widely regarded as a robust benchmark for testing both reasoning and coding proficiency [14].

4.1.2 Problems Selection

The automated evaluation aims to systematically assess the accuracy of SLMs, such as PHI-4 14B, LLAMA 3.2 3B, GEMMA 2 9B and DEEPSEEK-R1 14B, by evaluating their performance on 280 programming problems sourced from Codeforces API. Codeforces categorizes problem difficulty using an ELO-based rating system, ranging from 800 to 3000+. Given the relatively small size of these models and the inherent difficulty of Codeforces problems across rating levels, we carefully selected 20 problems from ratings 800 to 2100, prioritizing those with the highest number of successful submissions based on the solved count metric. This approach ensures a balanced and representative evaluation across varying difficulty levels. The problems used for this analysis were selected on Dec/15/2024.

These problems exhibit a broad range of token counts, as shown in Figure 4.1, with document requirements varying from 149 to 1,117 tokens. However, most requirements fall within the 227–459 token range. Additionally, these problems span a wide range of topics, as illustrated by the tag distribution in Figure 4.2, which highlights the 20 most frequently occurring topics. Since most problems are associated with multiple topics, the total tag count exceeds the number of problems. Notably, the “implementation” tag appears most frequently, as it encompasses problems where the core idea is straightforward, but the actual coding can be challenging. According to the Codeforces community, such problems often involve lengthy code, a higher likelihood of bugs, and numerous off-by-one errors [21].

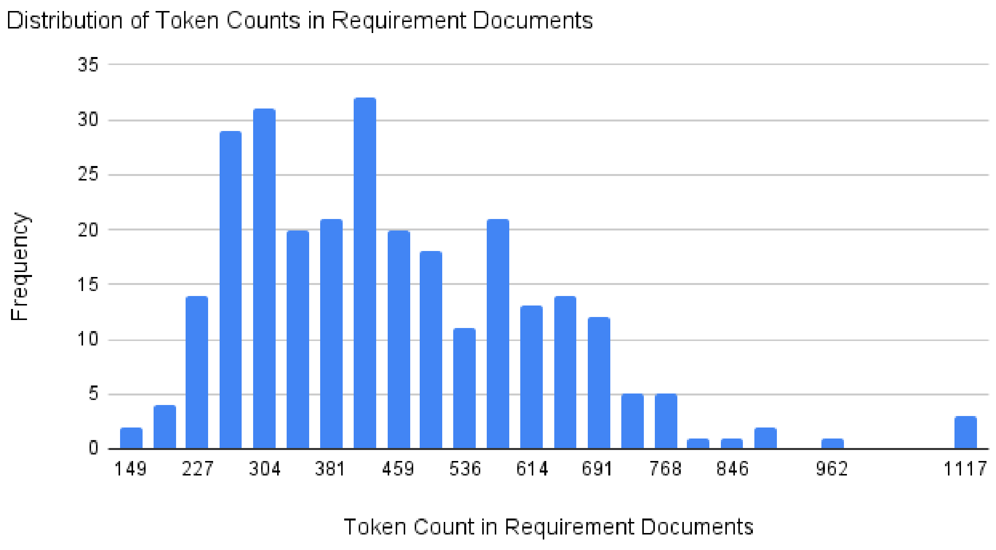


Figure 4.1: Distribution of Token Counts in Requirement Documents.

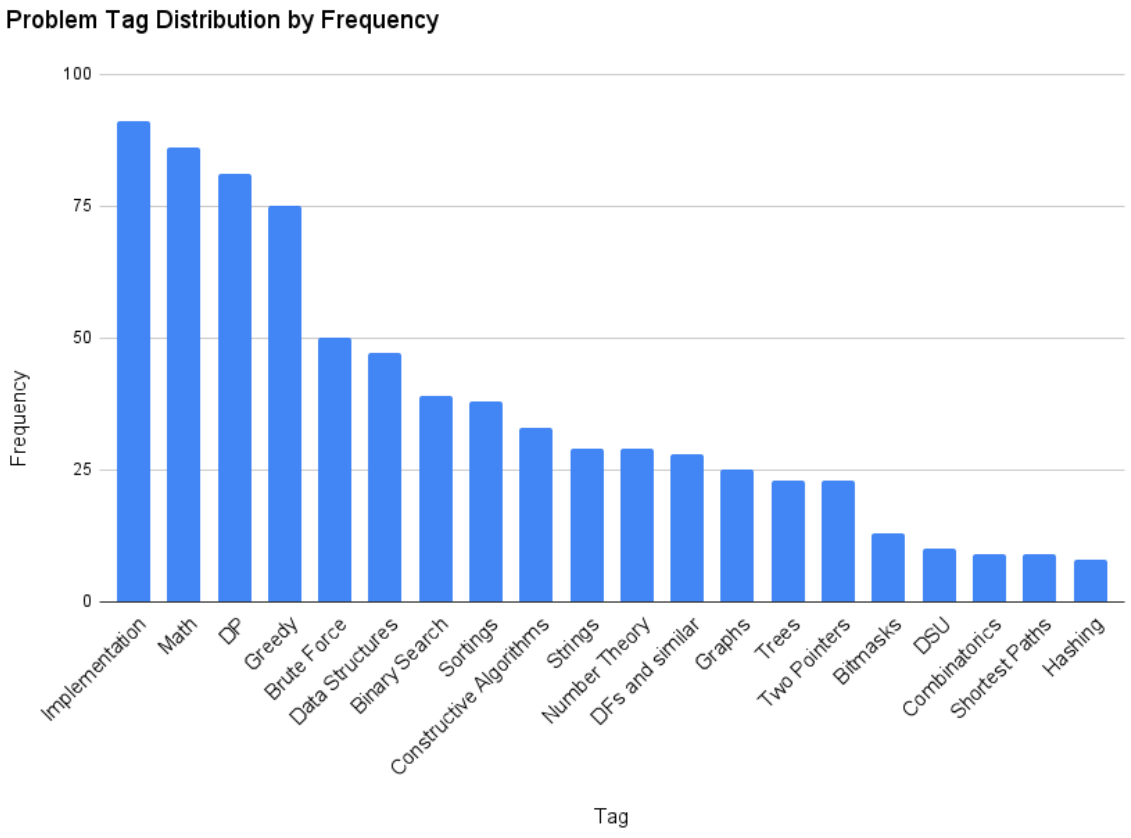


Figure 4.2: Problem Tag Distribution by Frequency.

4.1.3 Small Language Models Selection

The SLMs selected for evaluation in this study were chosen based on their performance and impact in the AI field. We utilized tools such as LLM Arena [6], a crowdsourced open platform for LM evaluations, to guide our selection based on their rankings. Additionally, we aimed to include one LM from different companies and architecture to ensure representation across different developers.

The small models selected for this study are LLAMA 3.2 3B [67], GEMMA 2 9B [33], PHI-4 14B [68] and DEEPSEEK-R1 14B [38], all of which are well-known language models suitable for tasks such as code generation, refactoring, and debugging, as outlined in Table 4.1. These models, with varying parameter sizes and context windows, are designed for use in tasks like automated code generation, refactoring, and debugging [1,38,66,79].

Table 4.1: Overview of SLMs: Settings, Parameters, and Release Information.

Model Name	Company	Number of Parameters	Context Window	Cutoff Date	Release Date
LLAMA 3.2 3B	Meta	3B	128K	December 2023	September 2024
GEMMA 2 9B	Google	9B	8k	-	June 2024
PHI-4 14B	Microsoft	14B	16K	June 2024	December 2024
DEEPSEEK-R1 14B	DeepSeek	14B	128K	-	January 2025

4.1.4 Prompt Design

To enhance the accuracy of problem-solving, we structured the prompt to include essential contextual information. This encompasses details about the persona, guidelines for approaching the solution, the designated programming language (Python), and a description of the problem statement format. These elements, combined with the problem statement itself, provide a comprehensive foundation to guide the problem-solving process effectively.

You are a highly skilled competitive programmer with 15 years of experience in the field. Your objective is to analyze the following problem statement and to produce a Python code solution that adheres to the requirements.

Guidelines for the solution: deliver only the Python code; Ensure the solution reads input via standard input and produces outputs results via standard output; If the solution

requires defining a function, ensure it is executed within the code; Avoid adding explanations, comments, or unnecessary text.
The Problem Statement includes a detailed description, input and output format, and examples to clarify requirements.
{*Problem Statement*}

4.1.5 Experiment Setup

The automated evaluation of SLMs was designed using problems from Codeforces to assess the performance of the PHI-4 14B, LLAMA 3.2 3B, GEMMA 2 9B and DEEPSEEK-R1 14B models. Although Codeforces offers an API for retrieving information about available problems, it does not provide full access to problem statements or allow direct submission of solutions. To overcome this limitation, we developed an automated tool using SeleniumBase to extract problem statements directly from the Codeforces platform and submit the generated code through the platform's interface. The dataset was curated in December 2024, with problems selected based on the number of solutions submitted. According to Codeforces' terms of use [18], web scraping is not prohibited.

To ensure a fair and consistent evaluation of the models, all executions were performed locally using the Ollama framework [75] in Python, running on an NVIDIA GeForce RTX 3060 GPU with 12GB of VRAM (February 2025). This hardware configuration was chosen to balance accessibility with computational efficiency. Each model was initialized by specifying its name and base URL, adhering to the default configurations provided by the LangChain Ollama API [55]. This approach ensures uniformity in execution settings, minimizing variability introduced by different model configurations or custom tuning.

Ollama was used to run the models locally, while SeleniumBase handled the automated retrieval of Codeforces problem statements and submission of solutions. Specifically, each problem was tested three times for each model, and the resulting code solutions were consolidated into a single CSV file. These solutions were then automatically submitted to Codeforces for correctness evaluation, and the returned verdicts were captured in a second CSV file. Together, these two CSV files formed the backbone of the exploratory data analysis.

This entire process, including the extraction of problem statements, the generation of solutions by the SLM, and the evaluation of the correctness on the Codeforces platform, was fully automated. A schematic overview is provided in Figure 4.3.

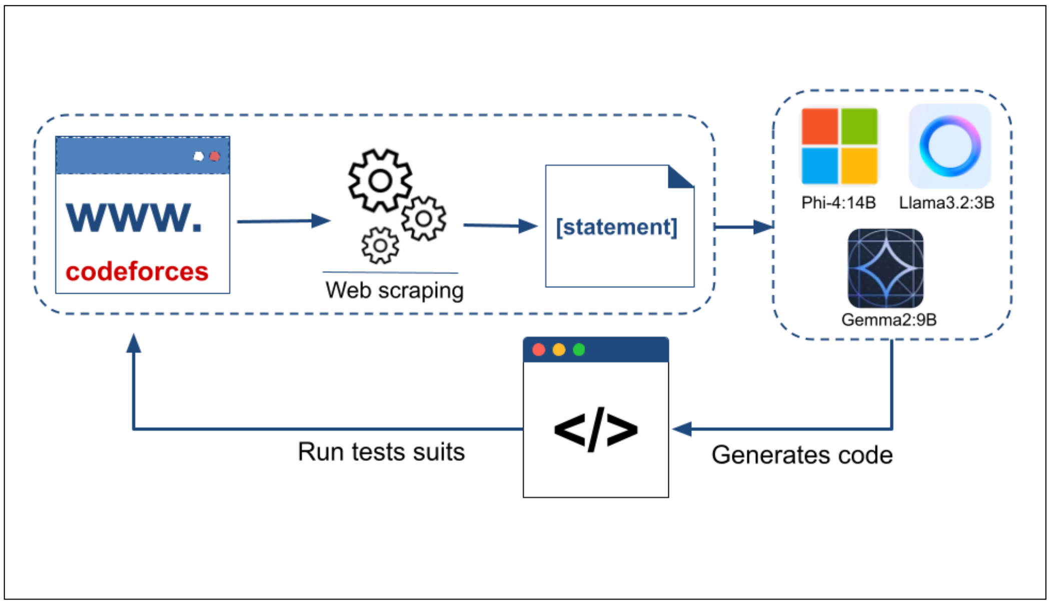


Figure 4.3: Setup of Automated Evaluation of SLMs.

4.1.6 Pilot Study

To refine the process of extracting problem statements, submitting solutions, and interacting with Codeforces, we conducted a step-by-step experiment using a sample of 10 problems out of the 280 selected on LLAMA 3.2 3B. During this experiment, we observed that the problem statements extracted from the platform sometimes contained descriptive terms for mathematical symbols, such as "le" for "$\leq$" and "ge" for "$\geq$". Since our goal was to maintain the integrity of the statements exactly as they appeared on Codeforces, we implemented a correction mechanism to automatically replace these textual representations with the appropriate mathematical symbols.

4.1.7 Experiment Phases

Building on the insights gained from the pilot experiment, the automated tool was scaled up to handle 280 problems, with 20 problems selected from each rating level. Additionally, three more models, PHI-4 14B, GEMMA 2 9B and DEEPSEEK-R1 14B, were incorporated into the evaluation process. This expanded phase aimed to systematically assess the performance of smaller models across a diverse set of problem difficulties. The experiment focused on key performance metrics such as accuracy and problem-solving consistency. Due to submission constraints on Codeforces, the experiment was organized by rating levels for each model, allowing for a structured and efficient testing approach.

4.2 Definition

In this section, we define the objective of our study using the **GQM** (Goal, Question, Metric) method [13]. Our goal is to **evaluate** SLMs, including PHI-4 14B, LLAMA 3.2 3B, GEMMA 2 9B and DEEPSEEK-R1 14B in solving programming problems sourced from Codeforces platform, **with the purpose** of raising implications of its use in the daily routine of programmers, regarding the correctness of the generated answers, **from the perspective of** researchers, **in the context of** automatic code generation.

Therefore, this study aims to provide insights into the practical implications of using SLMs in programmers' daily workflows, with a focus on solution correctness. To achieve this, we formulate and investigate three key research questions (RQs).

RQ$_4$ **To what extent SLMs as PHI-4 14B, LLAMA 3.2 3B and GEMMA 2 9B can answer programming assignments?**
To answer this question, the correct and incorrect responses provided by the Codeforces platform will be counted.

RQ$_5$ **What types of errors are most common in the responses generated by the SLMs?**
To address this question, we will examine and categorize the most frequent failures encountered in model outputs, including syntactic issues, logical missteps, and failures to satisfy problem constraints.

RQ$_6$ **How does the performance of SLMs vary across different programming topics?**
This question aims to assess the effectiveness of SLMs across a diverse set of programming topics, including graph algorithms, dynamic programming, sorting, and string manipulation. By analyzing performance variations, we can identify strengths and weaknesses in the models' problem-solving capabilities across different algorithmic domains.

4.3 Research Questions Results

In this chapter, we present the results obtained in the automated study carried out with 280 problems from Codeforces. The problems were analyzed with a main focus on the performance of the models.

4.3.1 RQ$_4$: To what extent SLMs as PHI-4 14B, LLAMA 3.2 3B, GEMMA 2 9B and DEEPSEEK-R1 14B can answer programming assignments?

In order to evaluate the model's answers to programming assignments, we selected 280 Codeforces problems spanning difficulty levels from 800 to 2100, with 20 problems at each level. To evaluate the model's performance, we employ the pass@k metric [14], which is widely used in evaluating code generated by models to quantify the probability that, among k solutions generated for a problem, at least one is correct. Because each problem was submitted to the model three times, we will present the performance metrics for pass@1, pass@2, and pass@3.

Table 4.2: LLAMA 3.2 3B, GEMMA 2 9B, PHI-4 14B and DEEPSEEK-R1 14B evaluations for Codeforces problems.

	pass@k		
Model	*pass@1*	*pass@2*	*pass@3*
LLAMA 3.2 3B	6.4%	10.4%	11.1%
GEMMA 2 9B	8.9%	10.0%	10.4%
DEEPSEEK-R1 14B	17.5%	22.9%	23.9%
PHI-4 14B	48.9%	58.6%	63.6%

According to Table 4.2, larger models such as PHI-4 14B and DEEPSEEK-R1 14B demonstrate significantly superior performance in solving programming problems. PHI-4 14B stands out as the best option, achieving an accuracy rate nearly three times higher than that of the second-place model, DEEPSEEK-R1 14B. The latter ranks second, with 17.50% in pass@1 and 23.93% in pass@3, outperforming the smaller models but still falling well behind PHI-4 14B. Meanwhile, GEMMA 2 9B and LLAMA 3.2 3B show similar performance, both with accuracy rates below 11% in pass@3, suggesting that smaller models may not be suitable for handling complex programming tasks.

4.3.2 RQ$_5$: What types of errors are most common in the responses generated by the SLMs?

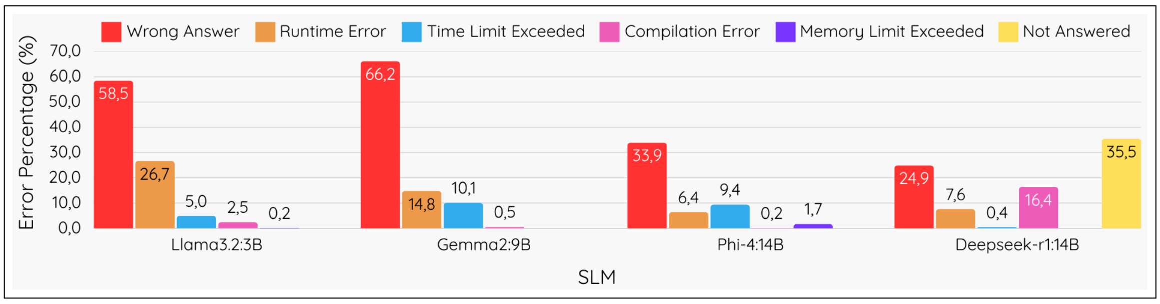


Figure 4.4: Percentage of submissions that resulted in an error, across 840 total submissions (280 problems, each submitted three times).

In this section, we examine the different types of errors that arise when the SLM fails to correctly solve a problem on the Codeforces platform. Codeforces evaluates each submission against a set of test cases and provides detailed feedback whenever a solution does not pass all tests. Common errors include wrong answers, Time Limit Exceeded (TLE), Memory Limit Exceeded (MLE), or runtime errors, each indicating a specific category of failure. By analyzing these messages, we gain insights into where the solution approach breaks down and how frequently each type of error occurs. Figure 4.4 offers a visual breakdown of the most prevalent errors, helping us understand patterns in unsuccessful submissions and guiding subsequent improvements to the SLM's performance.

As shown in the bar chart, Wrong Answer appears most often and happens when a solution fails to match the expected output for all test cases. The second most common issue is Runtime Error, which often results from using arrays or vectors with insufficient size or accessing invalid memory. Time Limit Exceeded and Memory Limit Exceeded also appear frequently, occurring when solutions run longer or use more memory than Codeforces allows.

The DeepSeek-R1 14B model is the only one that shows a "Not Answered" category because it often provides only a reasoning process without a final code solution. This happened in 35.5% of its submissions, which we classify as "Not Answered". In addition, DeepSeek-R1 14B frequently encountered compilation issues, with 16.4% of its responses resulting in a Compilation Error, as illustrated in the chart.

4.3.3 RQ$_6$: How does the performance of SLMs vary across different programming topics?

In this subsection, we examine the topics covered by the programming problems included in our study. Codeforces assigns one or more topic tags to each problem (e.g., "math," "greedy," "data structure"), making it easier to identify key concepts and skills tested. This tagging system provides valuable insight into which areas were most commonly addressed and helps us understand the range of problem-solving techniques the solution language models had to employ.

Table 4.3 lists the Top 10 most prevalent topics among the problems analyzed, alongside the percentage of questions successfully solved in each category. By focusing on these high-frequency topics, we gain a clearer picture of where the models excel and where further improvements may be needed.

Phi-4 14B (Phi) consistently performs better than the other three models, achieving noticeably higher success rates across every topic. This superiority is especially pronounced in categories such as Implementation, Math and Greedy, where even the second-best model falls significantly behind. Another key observation is that Strings appear to be a relatively strong suit for all models, though Phi-4 14B still outperforms the others.

In contrast, certain topics like Data Structures reveal clear weaknesses in the smaller models. Neither Llama 3.2 3B (Lla) nor Gemma 2 9B (Gem) manage any success there, while DeepSeek-R1 14B (DS) has limited, but non-zero, proficiency. Similarly, the Constructive Algorithms category remains challenging: Gem shows a modest edge over DS, yet both lag well behind Phi-4 14B. These patterns indicate that while model size and training strategies may offer advantages (as seen with DeepSeek-R1 14B and Phi-4 14B), more targeted refinements are needed for complex problem-solving topics.

Table 4.3: Performance of LLAMA 3.2 3B (Lla), GEMMA 2 9B (Gem), PHI-4 14B (Phi), and DEEPSEEK-R1 14B (DS) on the Top 10 topics in the selected Codeforces problems.

Topics	Lla	Gem	DS	Phi
Implementation	12.1%	15.8%	29.7%	67.0%
Math	5.4%	6.2%	11.2%	50.8%
Greedy	5.8%	4.0%	8.4%	51.1%
Brute Force	8.7%	8.7%	16.7%	50.0%
Sortings	3.5%	2.6%	12.3%	57.0%
Strings	27.6%	32.2%	39.1%	65.5%
Binary Search	0.0%	1.7%	6.8%	41.0%
Constructive Algorithms	1.0%	4.0%	4.0%	38.4%
Dynamic Programming	1.6%	2.1%	5.3%	32.1%
Data Structures	0.0%	0.0%	5.7%	28.4%

4.4 Discussion

Building upon the findings outlined in the preceding section, this section delves into various aspects related to the performance of the models. We investigate the self-consistency of each model and delve deeper to identify the difficulty levels of questions answered correctly and incorrectly. Furthermore, we analyzed the code proposed of each model to indicate the kind of code structures used and showcase how far a small mode can propose a complex model. We also analyzed the count of tests passed of the incorrect responses to indicate how far the response was from being correct.

4.4.1 Self-consistency of Small Language Models for Code Generation

In Section 4.3.1, we analyzed the rate of solved problems by SLMs using pass@k, a widely adopted metric in code generation evaluation. pass@k quantifies the probability that, among k generated solutions for a given problem, at least one is correct. While this metric provides insight into a model's accuracy across multiple attempts, it does not capture how consistently a model produces correct solutions across multiple submissions for the same problem.

To address this, we introduce the concept of consistency in solved problem rates. Since each problem is submitted three times, we analyze how many of the solved problems have at least two correct solutions out of those three attempts. Wang et al. and Xiong et al. [104,109] define consistency based on the equivalence of final answers, a criterion that is inadequate for open-ended tasks like code generation. Given that each problem was submitted three times, it is crucial to assess how consistently a model produces correct solutions across multiple attempts. To address this, we introduce Semantic Consistency (SC), which measures the proportion of problems for which a model generates at least 50% correct submissions out of three attempts, regardless of variations in syntax or implementation.

We define Semantic Consistency as follows:

$$\begin{array}{c}\text{Semantic}\\ \text{Consistency}\end{array} = \left(\frac{N_{\text{problems with }\geq 50\%\text{ of correct submissions}}}{N_{\text{solved problems}}}\right) \times 100\%$$

This metric provides a deeper understanding of model reliability, capturing the stability of correct responses across multiple submissions. By evaluating semantic consistency, we gain insights into how robust a model is in generating correct solutions consistently, rather than relying on a single correct response among multiple attempts.

Table 4.4 presents pass@3 and Semantic Consistency (SC), highlighting how reliably models generate repeatable correct solutions among the problems they successfully solved. All models exhibit high consistency, with SC values exceeding 60%, meaning that when a model solves a problem once, it is likely to solve it correctly multiple times. GEMMA 2 9B (86.2%) and PHI-4 14B (77.5%) show the strongest consistency, indicating that an overwhelming majority of their solved problems were correctly answered at least twice. DEEPSEEK-R1 14B (64.2%) and LLAMA 3.2 3B (61.3%) also maintain a moderate rate of repeatability, reinforcing their reliability in generating stable solutions.

These results emphasize that while pass@3 measures accuracy across all problems, Semantic Consistency, when analyzed within solved problems, shows that models tend to be stable and not merely generating correct answers by chance. Even models with lower accu-

Table 4.4: Comparison of pass@3 and Semantic Consistency, with an analysis of model consistency.

Models	pass@3	SC	Consistency Analysis
LLAMA 3.2 3B	11.1%	61.3%	**Moderate consistent.** A problem is correctly solved repeated in 61.3% of cases.
GEMMA 2 9B	10.4%	86.2%	**Highly consistent.** Among solved problems, 86.2% were correctly repeated.
DEEPSEEK-R1 14B	23.9%	64.2%	**Moderate consistent.** Once a problem is solved, it is answered correctly again in 64.2% of cases.
PHI-4 14B	63.6%	77.5%	**Highly consistent.** Nearly 77.5% of solved problems were answered correctly more than once.

racy, like LLAMA 3.2 3B and GEMMA 2 9B, still demonstrate a moderate ability to repeat correct answers when they do solve a problem.

4.4.2 Rating Levels of the Problems Answered

In Codeforces, problem difficulty is indicated by an ELO-based rating system, which starts around 800 for beginner-friendly questions and extends well beyond 3000 for expert-level challenges. In this study, we focused on ratings up to 2100, selecting 20 problems from each rating level. We set 2100 as the cutoff because this already represents a high level of difficulty, where problem-solving becomes significantly more challenging. Additionally, the overall success rate across models was low at this stage, making it a natural stopping point for our experiment.

Figure 4.5 reveals that PHI-4 14B consistently outperforms all other models across every

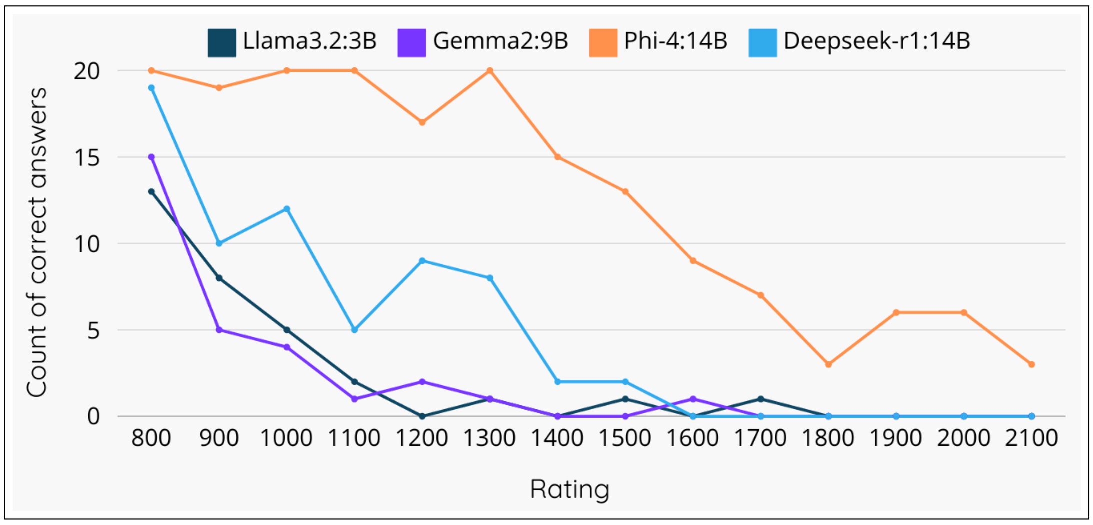


Figure 4.5: Performance of SLMs on Codeforces problems across difficulty levels. The x-axis represents problem difficulty levels (ratings from 800 to 2100), while the y-axis indicates the number of problems correctly solved out of 20 per level.

rating level considered. However, there is a clear downward trend in the number of solved problems as the difficulty rating increases, a pattern shared by all the small models. This observation underscores both the relative strengths of each SLM at lower to mid-range ratings and the significant challenges they encounter as problem complexity grows.

Each SLM demonstrates a unique range of problem-solving capabilities, aligning with broader performance trends. LLAMA 3.2 3B primarily succeeds in solving problems within the 800–1000 rating range, though it also manages to solve one problem each from the 1300, 1500, and 1700 levels. GEMMA 2 9B, on the other hand, covers a slightly broader spectrum, effectively handling problems rated 800–1200. However, its performance declines beyond this range, with only a single problem from the 1600 level being successfully solved. DEEPSEEK-R1 14B effectively solves problems within the 800–1500 rating range, demonstrating a steady problem-solving capability up to this level. PHI-4 14B, in contrast, exhibits the widest coverage, successfully addressing problems rated 800 through 2001. However, its performance declines sharply beyond the 1500 threshold, suggesting that while it is capable of solving higher-rated problems, its success rate decreases significantly as problem difficulty increases. This pattern highlights PHI-4 14B's ability to tackle a broad range of challenges, albeit with diminishing consistency at more advanced levels. Consequently, our assessment highlights both the distinct strengths of each model at lower to mid-range ratings and the challenges they face as problems become increasingly complex.

4.4.3 Costs

Running language models locally using Ollama can incur costs that depend largely on the type of GPU and the duration of its operation. In our study, we executed experiments on an NVIDIA GeForce RTX 3060 with 12GB VRAM, which has a TDP of approximately 170W [22]. We submitted 280 problems three times for each language model, totaling 840 submissions. The processing times varied considerably among the models: LLAMA 3.2 3B and GEMMA 2 9B each completed their runs in 0.5 hours, PHI-4 14B took 6 hours, and DEEPSEEK-R1 14B required 72 hours.

Considering the GPU's power consumption and an average energy rate of $0.1/kWh in Paraíba - Brazil [89], the estimated energy costs were approximately $0.01 for LLAMA 3.2 3B and GEMMA 2 9B, $0.1 for PHI-4 14B, and $1.2 for DEEPSEEK-R1 14B. These results not only illustrate the significant impact of model complexity on execution time, but also emphasize the feasibility of incorporating such models into everyday engineering workflows.

4.4.4 Analyzing the number of passed tests on unsolved problems on PHI-4 14B Model

Since the PHI-4 14B model has solved the most problems, we aim to analyze the problems it failed to solve in order to understand how close the model was to solving these challenges. The Codeforces API provides a field called "Passed Test Count," which indicates the number of tests passed by the solution. Figure 4.6 illustrates the number of tests passed by the proposed solution of the model for those problems it did not fully solve. This analysis helps to assess the model's performance and provides insights into its potential for solving unsolved problems in the future.

As previously mentioned, each problem was submitted three times, and we classify a submission as Incorrect if it fails to achieve at least two correct answers out of three attempts. The graph reveals a striking trend: a significant portion of the unsolved problems failed within the first test case. A deeper analysis of the data shows that, out of 306 incorrect submissions, 252 failed on the very first test case. This is particularly concerning, as the

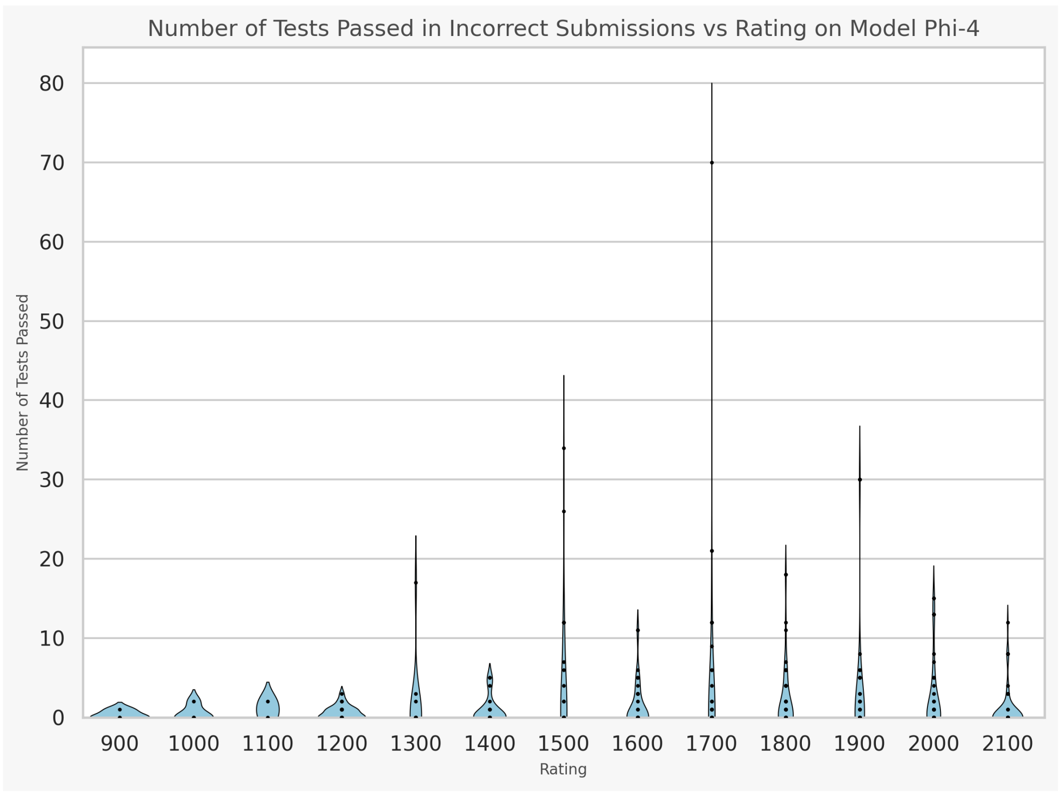


Figure 4.6: Number of Tests Passed in Incorrect Submissions vs Rating on Model PHI-4 14B.

initial test case is typically designed to mirror the problem statement's example, ensuring that even basic logic is correctly implemented.

On the other hand, there are cases where the model successfully passes a significant number of test cases but still fails to solve the problem completely. For instance, in problem "Checkposts" with rating 1700, which contains 137 test cases, the model managed to pass 70 before encountering a Runtime Error on test 71. This suggests that the generated solution covered a wide range of scenarios, but wasn't robust enough to fully solve the problem.

Another example is problem "Two Substrings" with rating 1500, where the model passed 34 out of 84 test cases before failing on test 35 with Wrong Answer. Interestingly, the model attempted three different submissions: one correct solution, one that reached 34 passed tests, and another that succeeded in just 7 cases. This indicates that while the model can be capable of generating solutions that perform well across multiple scenarios, it may still struggle with edge cases, ultimately leading to an incorrect classification.

4.4.5 Code Analysis of PHI-4 14B's Solutions

Since PHI-4 14B successfully solved most of the problems, we conducted an in-depth analysis of its generated code to identify the structures and commands it commonly employs as solutions. To achieve this, we used Python's Abstract Syntax Tree (AST) to systematically parse and quantify the occurrence of different syntactic constructs. This approach allows us to uncover patterns in the model's coding style and gain a deeper understanding of how PHI-4 14B structures its solutions. By examining these structural tendencies, we can assess the model's ability to write modular, efficient, and idiomatic Python code.

The analysis of the code structures used by the PHI-4 14B model is shown in Table 4.5 and it reveals a pattern strongly based on conditional control structures and iterative loops. The predominant use of "if" (628 occurrences) and "else" (250 occurrences) indicates that the model makes decisions based on logical checks, which is expected in problems involving multiple input cases. Additionally, the presence of 57 occurrences of "elif" reinforces the idea that many solutions require more complex branching, going beyond simple binary conditions. This suggests that the model adopts a structured approach to handling different scenarios within a problem.

The significant number of "for" loops (452 occurrences), along with a considerable amount of “while” loops (82 occurrences), demonstrates that the PHI-4 14B model makes extensive use of iterative structures to process data, execute repetitive operations, and navigate through collections. The dominance of for loops over while loops indicates that the solved problems generally involve iterations over known sequences (such as lists and dictionaries) rather than loops based on undefined conditions. This pattern is common in tasks involving list processing.

Regarding data structures, a moderate use of lists (193 occurrences) and dictionaries (28 occurrences) is observed, with a significantly lower frequency of sets (5 occurrences). This suggests that the model favors ordered and indexable structures, rather than sets, which are more suited for operations based on uniqueness. The considerable number of list comprehensions (55 occurrences) also indicates that the model is capable of producing more concise and efficient code in some situations, avoiding explicit loops when appropriate.

Finally, the analysis of function declarations and imports shows that the model structures its code in a modular manner. The presence of 330 function definitions (def) highlights the organization of the code into reusable components, which is essential for solving more complex problems. Additionally, the use of imports (import and from ... import) in 204 occurrences suggests that the model frequently relies on external libraries to solve problems, which may indicate an efficient approach by leveraging pre-existing functionalities, such as numerical or string manipulation. This demonstrates that PHI-4 14B not only writes functional code but also employs good practices in modularity and code reuse.

4.4.6 Evaluating PHI-4 14B after Cutoff Date

Since the PHI-4 14B model has a cutoff date of June 2024, we selected a set of six problems released on Codeforces after this date to evaluate its performance. As these problems could not have been included in GPT-4’s training data, they serve as a crucial benchmark for assessing the model’s ability to generalize to unseen challenges. The selected problems were published in Codeforces contests held between August 2024 and February 2025. Because these problems are newly posted, their ELO ratings may not yet accurately reflect their true difficulty. To mitigate this uncertainty, we specifically chose the first problem from each contest, as these are typically the most attempted. The selected contests span difficulty divisions ranging from Div4 to Div2.

Table 4.5: Code Structure Count of PHI-4 14B's Solutions.

Code Structure	Usage Count
lambdas	4
sets	5
from_import_statements	26
dictionaries	28
list_comprehensions	55
elif_statements	57
while_loops	82
import_statements	178
lists	193
return_statements	195
else_statements	250
functions	330
for_loops	452
if_statements	628

Table 4.6: Evaluation of Problems Released after the Cutoff date of PHI-4 14B.

Number of Correct Answers	PHI-4 14B	Problems Divs
3 Correct Answers	2	Div4
2 Correct Answers	2	Div4, Div2
1 Correct Answer	0	-
0 Correct Answer	2	Div3, Div4

As shown in Table 4.6, the PHI-4 14B model demonstrated strong performance on Div4 problems, successfully solving all test cases in two instances and partially solving two others (2 correct answers). This suggests that the model effectively handles introductory challenges. Additionally, its ability to achieve 2 correct answers on a Div2 problem indicates potential for tackling more advanced tasks, though it still struggles to comprehensively address all test cases.

Conversely, the model failed entirely on two problems from Div3 and Div4, highlighting its limitations even at intermediate difficulty levels. This pattern suggests that while PHI-4 14B can reliably solve straightforward problems, its robustness diminishes as complexity increases.

4.5 Threats to Validity

The study on the performance of Small Language Models like LLAMA 3.2 3B, PHI-4 14B, DEEPSEEK-R1 14B and GEMMA 2 9B in generating code from natural language descriptions, while illuminating, faces some threats to its validity. These threats can be categorized into internal and external validity threats, alongside construct and conclusion validity concerns:

4.5.1 Internal Validity

Problem Selection Bias: The programming problems selected from Codeforces might not cover all possible types of programming challenges, or may favor certain problem-solving paradigms. This could skew the evaluation towards models that perform better on these specific types of problem.

Formatting Loss: The problems were submitted in a textual form, meaning the text was extracted from the platform and submitted into SLM. Therefore, formatting loss may occur. To ensure consistency, the author preserved the original structure and format of the problem as presented on the platform, making an effort not to add or remove any line break.

Evaluation Criteria: Regarding the assessments of submissions, we assume that the platform's answers are correct. This approach is based on the widespread use of the platform by various developers. Any failure or error in the correction of questions would likely be identified and reported by the community, allowing for timely correction of these issues.

Success Criteria: Due to the probabilistic nature of the models, a correct answer may be generated by chance in one of the three submissions, which could introduce bias into the results. To mitigate this, the success criterion is defined as self-consistency—requiring that the model correctly solves at least two out of three attempts, ensuring more reliable performance evaluation.

Complexity of Problem Requirement Documents: A notable observation is that many problems originate from the Codeforces platform, which exhibits a higher complexity level in its problem statements. To prevent this, we addressed many different topics of problems, as well as different levels of problems.

4.5.2 External Validity

Generalization to Real-world Programming: The programming problems used in the study might not accurately reflect the complexity and diversity of real-world programming tasks. Thus, the models' performance in this controlled setting may not directly translate to effectiveness in practical coding scenarios.

Evolution of Models: These Small Language Models are rapidly evolving, with newer versions being released frequently. The findings may quickly become outdated, limiting the generalization of the study's conclusions over time.

4.5.3 Construct Validity

Evaluation of Correctness: How correctness is defined and measured in solving programming problems can significantly affect the outcomes. How our metric for success does not comprehensively capture the quality of the code generated in terms of efficiency, readability, or adherence to best practices, it may not accurately reflect the models' true capabilities.

Chapter 5

Related work

The evaluation of LLMs' coding capabilities has been conducted through different approaches. The first approach establishes benchmarks using datasets specifically designed to measure the performance of LLMs, such as HumanEval, MBPP, APPS. The second approach utilizes coding competition and practice platforms like LeetCode and BeeCrowd to directly test these capabilities [43].

Table 5.1 lists the main studies that evaluate LLMs in generating code from natural language. This table summarizes the key topics covered in these works, including the LLMs evaluated, the dataset used for testing, the programming language of the generated code, count of multiple attempts at problem submission, the usage of agentic approach, usage of metamorphic testing, and the small fix on requirement document.

The most cited models are GPT-3.5, Codex, GPT-J, GPT-4, and a variety of specialized models like CodeBERT, GraphCodeBERT, and AlphaCode-C. The tests are conducted with different datasets such as HumanEval, LeetCode, and CodeXGLUE, which include competitive programming problems and coding challenges in various languages like Python and Java. The research varies in terms of the number of tasks, attempts, and prompt styles used, providing a comprehensive view of the effectiveness of LLMs in automated code production.

Additionally, the table highlights the lack of metamorphic testing, an essential technique for evaluating the robustness and consistency of the models. None of the approaches mentioned use modifications to the requirements documents (RDs) to test the models' ability to adapt in different scenarios or fine-tune during execution, which may limit the generalization and accuracy of the models in real-world situations. Most of the studies also do not adopt an

Table 5.1: Summary of related work on LLMs coding generation task from natural language.

Paper	Year	Testing Dataset	LLMs	Code Generated Language	#Tasks	#Prompt Style	#Attempts	Metamorphic Test	Small Fix on RD	Agentic Aproach	Requirement Document Language
[14]	2021	HumanEval	GPT-3, Codex, GPT-J	Python	164	1	100	✗	✗	✗	English
[117]	2022	CodeXGLUE	CodeBERT, GraphCodeBERT, ContraCode, CodeGPT, CodeT5, CodeTrans, CoTexT, PLBART	Java	2000	1	5	✗	✗	✗	English
[110]	2022	HumanEval	Codex, GPT-J, GPT-Neo, CodeParrot, PolyCoder	Python	164	1	100	✗	✗	✗	English
[87]	2023	HumanEval, MBPP	GPT-4, Codex, Reflexion, Parsel, MetaGPT, CODE-T, CODE-T-Iter, LEVER + Codex, Reviewer + Codex002, MBR-Exec, AlphaCode-C	Python	664	3	1	✗	✗	✗	English
[93]	2023	Customized data science	GPT-3.5	Python	10	1	1	✗	✗	✗	English
[84]	2023	Customized Javascript	GPT-3.5	Javascript	1	1	1	✗	✗	✗	English
[85]	2023	Customized from LeetCode	GPT-3.5	-	128	1	1	✗	✗	✗	English
[62]	2023	CodeXGlue	GPT-3.5	Java	100	4	5	✗	✗	✗	English
[112]	2023	APPS	GPT-3.5	Python	4485	1	5	✗	✗	✗	English
[97]	2023	LeetCode	GPT-3.5, Claude, Spark, BingAI	Python	45	1	1	✗	✗	✗	English
[2]	2024	HumanEval, LeetCode, Codeforces	GPT-4, GPT-3.5	Python	331	1	100	✗	✗	✗	English
[34]	2024	HumanEval, MBPP	LLaMA 3	Python	1138	1	1	✗	✗	✗	English
[3]	2024	HumanEval, MBPP, APPS	GPT-4, CLAUDE 3 SONNET, GEMINI PRO 1.0, Gemini Ultra 1.0, Gemini Pro 1.5	Python	6138	1	1	✗	✗	✗	English
[98]	2024	HumanEval, Natural2Code	GPT-4, GPT-3.5, Gemini Ultra 1.0, Gemini Pro 1.0	Python	339	1	1	✗	✗	✗	English
[42]	2024	Customized from LeetCode, GeeksforGeeks	GPT-4, GPT-3.5, Claude 2, Gemini Pro 1.0, Gemini Ultra 1.0, Code Llama, Llama 2	Python	115	4	5	✗	✗	✗	English
[50]	2024	Stack Overflow	GPT-3.5	-	517	1	1	✗	✗	✗	English
[26]	2024	ClassEval	GPT-4, GPT-3.5, WizardCoder, Instruct-StarCoder, SantaCoder, Instruct-CodeGen, CodeGeeX, InCoder, Vicuna, ChatGLM, PolyCoder	Python	100	3	5	✗	✗	✗	English
[25]	2024	HumanEval, MBPP, APPS, CoderEval	GPT-4, GPT-3.5, AlphaCode, Incoder, CodeGeeX, StarCoder, CodeGen, PaLM Coder, CodeX, CodeX (175B) + CodeT, CodeLlama	Python	1821	3	1	✗	✗	✓	English
[49]	2024	HumanEval, HumanEval-X, MBPP-Sanitized, MBPP-ET, HumanEval-ET	AlphaCod, Incoder, CodeGeeX, CodeGen-Mono, PaLM Coder, code-davinci-002	Python, Java, Javascript, Go	1411	3	10	✗	✗	✓	English
[44]	2024	APPS, CodeF	GPT-3.5 Self-planning, GPT-3.5 SCOT, GPT-3.5 SCOT&KareCoder	Python	2023	1	10	✗	✗	✓	English
[73]	2024	ParEval	GPT-4, GPT-3.5, CodeLlama-7B, CodeLlama-13B, CodeLlama-34B, StarCoderBase, Phind-CodeLlama-V2	C++,	420	1	20	✗	✗	✗	English
[107]	2024	Customized from development team: project glossary, vision and scope, use cases	Tailored GPT-3.5	Java	1	1	1	✗	✗	✗	English

agent-based approach, where the model interacts proactively to refine or improve the generated code based on continuous feedback, which could enhance the efficiency and quality of the solutions generated.

Finally, the language of the requirements documents in all studies is predominantly English, reflecting the dominance of the language in LLM research for programming tasks. This may be a limitation, especially in contexts where technical documentation is not necessarily in English, or where other programming languages and frameworks are more prevalent. The use of English as the standard for input data may also affect the applicability of the models in different software development contexts, raising questions about the adaptability and transferability of models to other languages and cultures in automated coding.

In their study, Hou and Ji [43] evaluated the performance of LLMs such as GPT-4, Claude 2, LLaMA 2, GPT-3.5, Gemini Ultra, Gemini Pro, and Code LLaMA in code generation tasks using competitive programming platforms like LeetCode and GeeksforGeeks, with problems across three difficulty levels and allowing up to five attempts. The experiment utilized six different prompt strategies, including main approaches like Repeated Prompt, where the task is repeatedly presented to the LLM; Multiway Prompt, where five different solutions are generated; Feedback Prompt, where the error message from the previous attempt is used as input; and Feedback CI Prompt, where GPT-4 evaluates the test cases using the code interpreter (CI). In contrast, our study evaluates LLMs like GPT-4, CLAUDE 3 SONNET, LLAMA 3, and GEMINI PRO 1.0 on code generation tasks for competitive programming platforms such as LeetCode and BeeCrowd, with a maximum of three attempts, also using problems across three difficulty levels. We employed the Feedback Prompt strategy and expanded upon this research by categorizing the problems, analyzing errors raised during failed attempts, addressing potential data leakage, making a manual analysis of the requirement document to rephrase problems that are not solved by ambiguous statement and publishing the dataset used.

Yan et al. [112] conducted a study to evaluate the performance of ChatGPT (GPT-3.5), GPT-Neo, CodeRL on code generation from tasks of levels Introductory, Interview and Competition. They used APPS dataset to assess the effectiveness of GPT-3.5, and compared the solution generated by the LLMs and a ground-truth solution. Besides that, they evaluated the code quality in terms of "code cleanness". However, this research employs a prompt strategy that provides the test cases used to evaluate the problems directly to the LLM, which may result in biased outcomes. Additionally, the researchers do not delve deeply into the categorization of the problems, nor do they address the issue of data leakage, even though the dataset used may have been part of GPT-3.5's training data.

Kabir et al. [50] analyzed GPT-3.5's responses to 517 Stack Overflow questions to assess the correctness, consistency, conciseness, and comprehensiveness of the model's responses using a manual analysis. The Stack Overflow's questions used on the study are selected based on sub-categories such as Conceptual, How-to, Debugging, Popularity and Recency. This research showed that 52% of them contain incorrect information, 78% are inconsistent from human answers, 35% lack comprehensiveness, and 77% contain redundant, irrelevant, or unnecessary information. In contrast to this study, our research utilizes a dataset of competition problems from LeetCode and BeeCrowd, offering a more structured and controlled approach to analyzing the correctness of the answers provided by the LLMs.

In the work of Sakib et al. [85], is evaluated the code correctness of solutions generated by GPT-3.5 for LeetCode problems in two attempts. The problems range across three difficulty levels and address various topics such as Hash Table, Math, Two Pointers, Arrays, etc. Additionally, is discussed on the study the runtime and memory usage of the solutions proposed by the LLM. On our study, it was used the same strategy of prompt but using three attempts to submit the problems, we also expanded upon this research by analyzing errors raised during failed attempts, addressing potential data leakage, making a manual analysis of the requirement document to rephrase problems that are not solved by ambiguous statement and publishing the dataset used.

Dong et al. [25] presents an approach based on agents' communication. They created a self-collaboration framework for code generation employing LLMs. In this scenario, multiple LLM agents act as distinct "experts", each responsible for a specific subtask within a complex task, they collaborate and interact, so that different roles form a virtual team to facilitate each other's work. As a result, the virtual team addresses code generation tasks collaboratively without the need for human intervention. Using this methodology, they assemble a team consisting of three LLM roles (analyst, coder, and tester) responsible for software development's analysis, coding, and testing stages. Experimental results indicate that self-collaboration code generation relatively improves 29.9–47.1% compared to the base

LLM agent.

Another way to explore agentic approach was presented by Jiang et al. [49], they introduced a planning stage into code generation to help the LLM understand complex intent and reduce the challenges of problem-solving. In their work, it was proposed a self-planning code generation approach, which consists of two phases, namely planning phase and implementation phase. Using this method, LLM plans out concise solution steps from the intent combined with few-shot prompting. Subsequently, in the implementation phase, the model generates code step by step, guided by the preceding solution steps. Experimental results show that self-planning code generation achieves a improvement of up to 25.4% compared to direct code generation, and up to 11.9% compared to Chain-of-Thought of code generation.

Most of the studies analyze coding generation task on iterative coding tasks, using datasets like HumanEval or APPS. Going contrary to that, Nichols et al. [73] performed an experiment to study the capabilities of state-of-the-art language models to generate parallel code. In order to evaluate LLMs, they created a dataset, ParEval, consisting of prompts that represent 420 different coding tasks related to scientific and parallel computing. In the paper, they introduced two metrics for evaluating the runtime performance and scaling behavior of the generated parallel code. As a result, they found that LLMs are significantly worse at generating parallel code than they are at generating serial code.

Exploring a different format of input for LLMs, Bingyang Wei's study [107] presents a "Progressive Prompting" method that enables software engineers to interact with LLMs in a stepwise, iterative manner. This approach allows the LLM to process comprehensive project files, including glossaries, scope, and use cases, as input. Through Progressive Prompting, the LLM progressively addresses software development tasks, starting with interpreting the provided requirements to identify functional requirements. It then uses these requirements to create object-oriented models, followed by generating unit tests and code based on the developed designs. This study highlights the potential of integrating LLMs into the software development workflow, significantly improving both efficiency and quality.

Chapter 6

Conclusions

In this work, we conduct an in-depth analysis of the performance of several prominent LLMs such as GPT-4, GEMINI PRO 1.0, LLAMA 3, and CLAUDE 3 SONNET in generating code for programming challenges sourced from platforms like LeetCode and BeeCrowd. Additionally, we examine the performance of smaller LMs, including LLAMA 3.2 3B, GEMMA 2 9B, DEEPSEEK-R1 14B and PHI-4 14B, through an automated approach, with a particular emphasis on code generation tasks sourced from the Codeforces platform. This analysis sheds light on the practical potential and advancements of Language Model technologies, emphasizing their evolving role and influence in the field of Software Engineering.

In the analysis of Large Language Models, GPT-4 stands out with superior performance, achieving a 78% accuracy rate in providing correct responses. In contrast, the other models perform below 71% in terms of accuracy. The study also highlights a significant disparity in the models' ability to tackle problems from LeetCode and BeeCrowd platforms. Specifically, while 95% of the problems from LeetCode are answered correctly, only 31% of the BeeCrowd problems yield correct solutions. Despite demonstrating strong performance on LeetCode, this stark contrast emphasizes the models' weaknesses on BeeCrowd, indicating areas in code generation that require further refinement.

The observed discrepancy in accuracy between the platforms can be attributed to the distinct nature of their problem requirements. BeeCrowd's challenges are often designed for competitive programming scenarios, incorporating narrative elements with fictional and playful components. These challenges are intentionally crafted with a level of ambiguity, adding complexity and requiring the models to navigate through unclear or creatively constructed contexts. On the other hand, LeetCode's problems are typically characterized by straightforward, precise instructions, with a strong emphasis on assessing specific technical skills. The absence of unnecessary complexity in LeetCode problems allows for more direct evaluation, which likely contributes to the models' higher success rate on this platform.

The GPT-4 model achieved 100% accuracy on problems from the LeetCode platform, which is why no errors were recorded for this model. For the other models, across both LeetCode and BeeCrowd, the most frequently encountered error was "Wrong Answer" indicating that the generated code failed to pass all test cases.

On the BeeCrowd platform, which exhibited a higher overall number of errors, additional frequent issues emerged, such as "Time Limit Exceeded," where the code exceeded the allowed execution time, followed by "Runtime Error," indicating failures during execution due to factors like memory access violations or division by zero.

Regarding problem difficulty, the analyzed models exhibit a clear trend in their performance on the BeeCrowd platform. Easy problems have the highest success rate, followed by medium, with hard problems being the least correctly solved. However, this pattern contrasts with the results from LeetCode, where the models were able to correctly solve problems across all difficulty levels, demonstrating a more consistent performance regardless of complexity.

For the Small Language Models, the evaluation was conducted using problems from the Codeforces platform. Among the evaluated models, PHI-4 14B displayed superior performance, successfully solving 49% of the submitted problems, including more challenging tasks with ratings as high as 2000. In contrast, the smaller models, GEMMA 2 9B and LLAMA 3.2 3B, showed significantly lower performance, with correct solutions for only 8% of the problems. The LLAMA 3.2 3B model's solutions were mostly limited to problems with ratings ranging from 800 to 900, while GEMMA 2 9B managed to solve problems with ratings up to 1300. This highlights the varying degrees of capability among these models, with PHI-4 14B excelling at more complex problems, while the smaller models demonstrated a more constrained range of successful problem-solving.

For smaller models, the most frequently encountered error across all three analyzed models is "Wrong Answer" indicating that the generated solution fails to pass all test cases. This is followed by "Runtime Error" which often stems from issues such as defining an array with insufficient capacity or attempting to access an invalid memory location, highlighting potential weaknesses in memory management and problem constraints handling.

It is important to note that Codeforces problems are typically designed for competitive programming, often with greater complexity and intricacy. Despite these challenges, this analysis aims to provide insights into how different sizes of small models perform in handling such problems, showcasing the varied capabilities of these models in a demanding competitive environment.

For the Codeforces problems, we use the ELO rating system to determine difficulty levels. A clear trend emerges across all three analyzed small models: as the problem rating increases, the number of correct solutions declines. This drop becomes particularly steep beyond the 1500 rating, where PHI-4 14B's success rate falls from 11 correct solutions to just 4, with no recovery beyond that point.

Our analysis highlights significant performance disparities among LLMs in code generation tasks across different platforms. GPT-4 demonstrated outstanding accuracy, particularly on LeetCode, while other models struggled, especially with BeeCrowd problems, which feature greater ambiguity and complexity. Among SLMs, PHI-4 14B showed the highest success rate, particularly in handling more challenging Codeforces problems, whereas the smaller models, GEMMA 2 9B and LLAMA 3.2 3B, exhibited limited problem-solving capabilities. Additionally, a consistent trend was observed across all models: as problem difficulty increased, accuracy declined, with error patterns revealing critical areas for improvement, such as handling edge cases, optimizing execution time, and improving memory management. These findings emphasize the evolving strengths and limitations of current LLMs in software engineering tasks, underscoring the need for further advancements in model robustness and adaptability.

6.1 Implications of Utilizing LLMs for Code Generation

The utilization of LLMs such as GPT-4, GEMINI PRO 1.0, LLAMA 3, CLAUDE 3 SONNET in software engineering can offer several advantages but also comes with cons. For software engineers, these models can be advantageous as they can assist in code generation, provide solutions to programming problems, and speed up some aspects of software development.

However, as evidenced, all the models do not achieve 100% accuracy, so programmers should not see them as substitutes. They may not always produce accurate or optimal code, and there could be challenges related to code quality, performance optimization, and security considerations. Software engineers should not rely solely on LLM-generated code but rather use it as a supplement to their expertise and judgment.

For students, there is the advantage of having an assistant to explain content and clarify doubts, but there are also disadvantages, as the models evaluated here may provide incorrect answers and the student may not be equipped to detect the error.

In summary, while LLMs can be valuable tools in software engineering for tasks such as code generation and problem-solving, software engineers must approach their usage with caution and critical thinking to evaluate the generated response and not use it without proper testing.

6.2 Study Limitations and Future Directions

Despite revolutionizing the code generation landscape and delivering impressive performance across a variety of tasks, LMs still face significant challenges that limit their applicability in real-world software development. Many of these limitations arise from the disconnect between academic benchmarks and practical coding environments. While models tend to perform well on controlled datasets, their effectiveness often declines in production settings due to factors such as ambiguous problem descriptions, computational constraints, and the lack of deeper contextual reasoning. Overcoming these challenges requires advances in model robustness, improved handling of edge cases, and better alignment with industry requirements to ensure reliable and maintainable code in realistic development scenarios.

A key direction for addressing this gap lies in expanding the scope of code generation beyond isolated function-level tasks to repository- and system-level challenges. In day-to-day development, programmers frequently face a broad spectrum of complex problems involving interdependent modules, architectural constraints, and domain-specific logic [58, 118]. Although LMs have shown notable success in generating self-contained code snippets, they continue to struggle with unseen problems that demand cross-file reasoning and high-level design decisions. Tackling such scenarios requires language models to evolve into more capable problem solvers, able to reason not only about algorithms, but also about code organization, scalability, and long-term maintainability.

As a continuation of this research, we conducted a deeper evaluation focused on SLMs, published in a follow-up study [91]. This work benchmarked five open SLMs on a diverse set of Codeforces problems, extending the analysis to include multi-language experiments and qualitative error inspections. Building on these insights, future work will explore the use of agent-based workflows with SLMs. Recent studies have demonstrated the effectiveness of such approaches in tasks like test smell detection and automated refactoring [65], suggesting that multi-agent collaboration could enhance the reasoning capabilities of SLMs—particularly when tackling complex, multistep programming challenges in competitive and professional environments.

Bibliography

[1] Marah Abdin, Jyoti Aneja, Harkirat Behl, Sébastien Bubeck, Ronen Eldan, Suriya Gunasekar, Michael Harrison, Russell J Hewett, Mojan Javaheripi, Piero Kauffmann, et al. Phi-4 technical report. *arXiv preprint arXiv:2412.08905*, 2024.

[2] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. GPT-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[3] Anthropic. The Claude 3 Model Family: Opus, Sonnet, Haiku. `https://paperswithcode.com/paper/the-claude-3-model-family-opus-sonnet-haiku`, 2024. Accessed on: March 10, 2025.

[4] Appinventiv. How much does it cost to hire software developers? `https://appinventiv.com/blog/hire-software-developer/`, 2024. Accessed on: March 10, 2025.

[5] Leonhard Applis, Annibale Panichella, and Ruben Marang. Searching for quality: Genetic algorithms and metamorphic testing for software engineering ML. In *Proceedings Of The Genetic And Evolutionary Computation Conference*, pages 1490–1498, 2023.

[6] LLM arena. LMSYS chatbot arena leaderboard. `https://lmarena.ai/`, 2024. Accessed on: March 10, 2025.

[7] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

[8] Satanjeev Banerjee and Alon Lavie. METEOR: An automatic metric for mt evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, pages 65–72, 2005.

[9] BeeCrowd. BeeCrowd platform. `https://BeeCrowd.com.br/`, 2024. Accessed on: March 10, 2025.

[10] BeeCrowd. Categories from BeeCrowd platform. `https://www.beecrowd.com.br/judge/en/categories`, 2024. Accessed on: March 10, 2025.

[11] BeeCrowd. FAQS Judge - BeeCrowd. `https://www.beecrowd.com.br/judge/en/faqs`, 2024. Accessed on: March 10, 2025.

[12] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155, 2003.

[13] Victor R Basili1 Gianluigi Caldiera and H Dieter Rombach. The goal question metric approach. *Encyclopedia of software engineering*, pages 528–532, 1994.

[14] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating Large Language Models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[15] Stanley F Chen, Douglas Beeferman, and Roni Rosenfeld. Evaluation metrics for language models. *Journal contribution*, 1998.

[16] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, TH Tse, and Zhi Quan Zhou. Metamorphic testing: A review of challenges and opportunities. *ACM Computing Surveys (CSUR)*, 51(1):1–27, 2018.

[17] Codeforces. Codeforces platform. `https://codeforces.com/`, 2024. Accessed on: March 10, 2025.

[18] Codeforces. Terms of Use of Codeforces. `https://codeforces.com/terms`, 2024. Accessed on: March 10, 2025.

[19] CodingScape. The Most Powerful Large Language Models (LLMs) in 2023. `https://codingscape.com/blog/most-powerful-llms-large-language-models`, 2023. Accessed on: March 10, 2025.

[20] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *arXiv preprint arXiv:1103.0398*, 2011.

[21] Codeforces Community. Implementation Tag meaning. `https://codeforces.com/blog/entry/100832`, 2022. Accessed on: March 10, 2025.

[22] NVIDIA Corporation. Geforce RTX 3060 e RTX 3060 Ti. `https://www.nvidia.com/pt-br/geforce/graphics-cards/30-series/rtx-3060-3060ti/`, 2025. Accessed on: March 10, 2025.

[23] Badhan Chandra Das, M Hadi Amini, and Yanzhao Wu. Security and privacy challenges of large language models: A survey. *ACM Computing Surveys*, 57(6):1–39, 2025.

[24] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, pages 4171–4186, 2019.

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

[26] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. Evaluating large language models

in class-level code generation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[27] Ahmed El-Kishky, Alexander Wei, Andre Saraiva, Borys Minaev, Daniel Selsam, David Dohan, Francis Song, Hunter Lightman, Ignasi Clavera, Jakub Pachocki, et al. Competitive programming with large reasoning models. *arXiv preprint arXiv:2502.06807*, 2025.

[28] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. CodeBERT: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[29] Richard E Fikes and Nils J Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1971.

[30] William A Gale and Geoffrey Sampson. Good-turing frequency estimation without tears. *Journal of quantitative linguistics*, 2(3):217–237, 1995.

[31] Jianfeng Gao and Chin-Yew Lin. Introduction to the special issue on statistical language modeling, 2004.

[32] Gartner. Gartner forecasts worldwide it spending to grow 7.5% in 2024. `https://www.gartner.com/en/newsroom/press-releases/2024-07-16-gartner-forecasts-worldwide-it-spending-\protect\penalty\z@to-grow-7-point-5-percent-in-2024`, 2024. Accessed on: March 10, 2025.

[33] Google. Gemma 2:9B. `https://ollama.com/library/gemma2`, 2024. Accessed on: March 10, 2025.

[34] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The Llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[35] Cordell Green. Theorem proving by resolution as a basis for question-answering systems. *Machine intelligence*, 4:183–205, 1969.

[36] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices*, 46(1):317–330, 2011.

[37] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.

[38] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

[39] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. Measuring coding challenge competence with APPs. *arXiv preprint arXiv:2105.09938*, 2021.

[40] Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*, 2021.

[41] Robert M Hierons, Kirill Bogdanov, Jonathan P Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul Krause, et al. Using formal specifications to support testing. *ACM Computing Surveys (CSUR)*, 41(2):1–76, 2009.

[42] Wenpin Hou and Zhicheng Ji. A systematic evaluation of large language models for generating programming code. *arXiv e-prints*, pages arXiv–2403, 2024.

[43] Wenpin Hou and Zhicheng Ji. Comparing large language models and human programmers for generating programming code. *Advanced Science*, 12(8):2412279, 2025.

[44] Tao Huang, Zhihong Sun, Zhi Jin, Ge Li, and Chen Lyu. Knowledge-aware code generation with large language models. In *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*, pages 52–63, 2024.

[45] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holis-

tic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.

[46] Fred Jelinek, Robert L Mercer, Lalit R Bahl, and James K Baker. Perplexity—a measure of the difficulty of speech recognition tasks. *The Journal of the Acoustical Society of America*, 62(S1):S63–S63, 1977.

[47] JetBrains. The state of developer ecosystem 2024. `https://www.jetbrains.com/lp/devecosystem-data-playground`, 2024. Accessed on: March 10, 2025.

[48] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on Large Language Models for code generation. *arXiv preprint arXiv:2406.00515*, 2024.

[49] Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. Self-planning code generation with Large Language Models. *ACM Transactions on Software Engineering and Methodology*, 33(7):1–30, 2024.

[50] Samia Kabir, David N Udo-Imeh, Bonan Kou, and Tianyi Zhang. Is stack overflow obsolete? an empirical study of the characteristics of ChatGPT answers to Stack Overflow questions. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, pages 1–17, 2024.

[51] Katikapalli Subramanyam Kalyan. A survey of GPT-3 family large language models including ChatGPT and GPT-4. *Natural Language Processing Journal*, 6:100048, 2024.

[52] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

[53] Slava Katz. Estimation of probabilities from sparse data for the language model component of a speech recognizer. *IEEE transactions on acoustics, speech, and signal processing*, 35(3):400–401, 2003.

[54] Stefan Kombrink, Tomas Mikolov, Martin Karafiát, and Lukás Burget. Recurrent neural network based language modeling in meeting recognition. In *Interspeech*, volume 11, pages 2877–2880, 2011.

[55] LangChain API. Ollama LLM. `https://api.python.langchain.com/en/latest/ollama/llms/langchain_ollama.llms.OllamaLLM.html`, 2025. Accessed on: March 10, 2025.

[56] Leetcode. Leetcode platform. `https://leetcode.com/`, 2024. Accessed on: March 10, 2025.

[57] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461*, 2019.

[58] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, 2022.

[59] Yujia Li, Yoonho Choi, Junyoung Wang, Vikranth Mehta, Maarten Bosma, Ivo Danihelka, Edward Grefenstette, Jakub Tomczak, and Oriol Vinyals. Competition-Level Code Generation with AlphaCode. arXiv preprint arXiv:2203.07814, 2022.

[60] Chin-Yew Lin. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, pages 74–81, 2004.

[61] Wang Ling, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiskỳ, Andrew Senior, Fumin Wang, and Phil Blunsom. Latent predictor networks for code generation. *arXiv preprint arXiv:1603.06744*, 2016.

[62] Chao Liu, Xuanlin Bao, Hongyu Zhang, Neng Zhang, Haibo Hu, Xiaohong Zhang, and Meng Yan. Improving ChatGPT prompt for code generation. *arXiv preprint arXiv:2305.08360*, 2023.

[63] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36:21558–21572, 2023.

[64] Michael R Lyu, Baishakhi Ray, Abhik Roychoudhury, Shin Hwei Tan, and Patanamon Thongtanunam. Automatic programming: Large language models and beyond. *ACM Transactions on Software Engineering and Methodology*, 2024.

[65] Rian Melo, Pedro Simões, Rohit Gheyi, Marcelo d'Amorim, Márcio Ribeiro, Gustavo Soares, Eduardo Almeida, and Elvys Soares. Agentic SLMs: Hunting Down Test Smells. *arXiv preprint arXiv:2504.07277*, 2025.

[66] Meta. Llama 3.2: Revolutionizing edge AI and vision with open, customizable models, 2024. Accessed on: March 10, 2025.

[67] Meta. Llama 3.2:3B. `https://ollama.com/library/llama3.2`, 2024. Accessed on: March 10, 2025.

[68] Microsoft. Phi-4:14B. `https://ollama.com/library/phi4:14b`, 2024. Accessed on: March 10, 2025.

[69] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[70] Tomas Mikolov, Martin Karafiát, Lukas Burget, Jan Cernockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Interspeech*, pages 1045–1048. Makuhari, 2010.

[71] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, 26, 2013.

[72] Raghav Mittal. What is an ELO Rating? `https://medium.com/purple-theory/what-is-elo-rating-c4eb7a9061e0`, 2020. Accessed on: March 10, 2025.

[73] Daniel Nichols, Joshua H Davis, Zhaojun Xie, Arjun Rajaram, and Abhinav Bhatele. Can large language models write parallel code? In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*, pages 281–294, 2024.

[74] Theo X Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. Is self-repair a silver bullet for code generation? *arXiv preprint arXiv:2306.09896*, 2023.

[75] Ollama. Ollama LLMs. `https://ollama.com/`, 2024. Accessed on: March 10, 2025.

[76] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.

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

[78] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer. Deep contextualized word representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2018*, pages 2227–2237, New Orleans, Louisiana, USA, June 2018. Association for Computational Linguistics.

[79] PromptHackers. Comparison of gemma 2:9b vs llama 3.2:3b, 2025. Accessed on: March 10, 2025.

[80] Maxim Rabinovich, Mitchell Stern, and Dan Klein. Abstract syntax networks for code generation and semantic parsing. *arXiv preprint arXiv:1704.07535*, 2017.

[81] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[82] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. CodeBLEU: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*, 2020.

[83] Ronald Rosenfeld. Two decades of statistical language modeling: Where do we go from here? *Proceedings of the IEEE*, 88(8):1270–1278, 2000.

[84] Ahmed R Sadik, Antonello Ceravola, Frank Joublin, and Jibesh Patra. Analysis of ChatGPT on source code. *arXiv preprint arXiv:2306.00597*, 2023.

[85] Fardin Ahsan Sakib, Saadat Hasan Khan, and AHM Rezaul Karim. Extending the frontier of ChatGPT: Code generation and debugging. In *2024 International Conference on Electrical, Computer and Energy Technologies (ICECET*, pages 1–6. IEEE, 2024.

[86] June Sallou, Thomas Durieux, and Annibale Panichella. Breaking the silence: the threats of using LLMs in software engineering. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, pages 102–106, 2024.

[87] Jiho Shin, Clark Tang, Tahmineh Mohati, Maleknaz Nayebi, Song Wang, and Hadi Hemmati. Prompt engineering or fine tuning: An empirical assessment of large language models in automated software engineering tasks. *arXiv preprint arXiv:2310.10508*, 2023.

[88] Craig Smith. What Large Models Cost You—There Is No Free AI Lunch, 2023. Accessed on: March 10, 2025.

[89] Portal Solar. Ranking of the most expensive energy tariffs in Brazil in 2024. `https://www.portalsolar.com.br/noticias/mercado/consumidor/confira-o-ranking-das-tarifas-de-energia-\protect\penalty\z@mais-caras-do-brasil-em-2024`, 2024. Accessed on: March 10, 2025.

[90] Débora Souza. Comparing Gemini Pro and GPT-3.5 in algorithmic problems. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pages 698–700, 2024.

[91] Débora Souza, Rohit Gheyi, Lucas Albuquerque, Gustavo Soares, and Márcio Ribeiro. Code Generation with Small Language Models: A Deep Evaluation on Codeforces. *arXiv preprint arXiv:2504.07343*, 2025.

[92] Débora Souza and Rohit Gheyi. Case study: using ChatGPT to solve programming problems. In *Extended Proceedings of the XIV Brazilian Software Conference: Theory and Practice*, pages 80–89, Porto Alegre, RS, Brazil, 2023. SBC. Awarded 1st place at CTIC 2023.

[93] Giriprasad Sridhara, Sourav Mazumdar, et al. ChatGPT: A study on its utility for ubiquitous software engineering tasks. *arXiv preprint arXiv:2305.16837*, 2023.

[94] Nature Editorial Staff. The hidden costs of AI: Why large models are more expensive than they seem, 2025. Accessed on: March 10, 2025.

[95] Wired Staff. Generative AI and climate change are on a collision course, 2024. Accessed on: March 10, 2025.

[96] Andreas Stolcke. Srilm-an extensible language modeling toolkit. In *Interspeech*, 2002.

[97] Haoran Su, Jun Ai, Dan Yu, and Hong Zhang. An evaluation method for large language models' code generation capability. In *2023 10th International Conference on Dependable Systems and Their Applications (DSA)*, pages 831–838. IEEE, 2023.

[98] Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, et al. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

[99] Gemma Team, Morgane Riviere, Shreya Pathak, Pier Giuseppe Sessa, Cassidy Hardin, Surya Bhupatiraju, Léonard Hussenot, Thomas Mesnard, Bobak Shahriari,

Alexandre Ramé, et al. Gemma 2: Improving open language models at a practical size, 2024. *URL https://arxiv. org/abs/2408.00118*, 1(3), 2024.

[100] Alan M Turing. Computing machinery and intelligence. *Mind*, LIX(236):433–460, 1950.

[101] Uptech. Software development costs: All factors explained. `https://www.uptech.team/blog/software-development-costs`, 2024. Accessed on: March 10, 2025.

[102] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[103] Richard J Waldinger and Richard CT Lee. PROW: A step toward automatic program writing. In *Proceedings of the 1st international joint conference on Artificial intelligence*, pages 241–252, 1969.

[104] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.

[105] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021.

[106] Yuepeng Wang, James Dong, Rushi Shah, and Isil Dillig. Synthesizing database programs for schema refactoring. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 286–300, 2019.

[107] Bingyang Wei. Requirements are all you need: From requirements to code with LLMs. In *2024 IEEE 32nd International Requirements Engineering Conference (RE)*, pages 416–422. IEEE, 2024.

[108] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, et al. Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682*, 2022.

[109] Miao Xiong, Zhiyuan Hu, Xinyang Lu, Yifei Li, Jie Fu, Junxian He, and Bryan Hooi. Can LLMs express their uncertainty? an empirical evaluation of confidence elicitation in LLMs. *arXiv preprint arXiv:2306.13063*, 2023.

[110] Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN international symposium on machine programming*, pages 1–10, 2022.

[111] Frank F Xu, Zhengbao Jiang, Pengcheng Yin, Bogdan Vasilescu, and Graham Neubig. Incorporating external knowledge through pre-training for natural language to code generation. *arXiv preprint arXiv:2004.09015*, 2020.

[112] Dapeng Yan, Zhipeng Gao, and Zhiming Liu. A closer look at different difficulty levels code generation abilities of ChatGPT. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1887–1898. IEEE, 2023.

[113] Pengcheng Yin and Graham Neubig. A syntactic neural model for general-purpose code generation. *arXiv preprint arXiv:1704.01696*, 2017.

[114] Pengcheng Yin and Graham Neubig. TRANX: A transition-based neural abstract syntax parser for semantic parsing and code generation. *arXiv preprint arXiv:1810.02720*, 2018.

[115] Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. Codereval: A benchmark of pragmatic code generation with generative pre-trained models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–12, 2024.

[116] Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Yongji Wang, and Jian-Guang Lou. Large language models meet NL2Code: A survey. *arXiv preprint arXiv:2212.09420*, 2022.

[117] Zhengran Zeng, Hanzhuo Tan, Haotian Zhang, Jing Li, Yuqun Zhang, and Lingming Zhang. An extensive study on pre-trained models for program understanding and generation. In *Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis*, pages 39–51, 2022.

[118] Jialu Zhang, De Li, John Charles Kolesar, Hanyuan Shi, and Ruzica Piskac. Automated feedback generation for competition-level code. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–13, 2022.

[119] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 1(2), 2023.