Proceedings of SEML 2025 Symposium: Machine Learning Theory and Applications
DOI: 10.54254/2755-2721/2025.TJ22242

Evaluating Large Language Models for Code Generation: A Comparative Study on Python, Java, and Swift

Boqiao Wan

The University of Melbourne, Melbourne, Australia
boqiaow@gmail.com

Abstract: With the development of artificial intelligence (AI), particularly in natural language processing and machine learning, AI applications in code generation, error correction, and programming assistance have become more common. However, differences in code generation capabilities among models influence their practical applicability in programming tasks. To investigate this issue, this study evaluates the performance of five state-of-the-art large language models (LLMs)—GPT-4o, OpenAI o1, OpenAI o1 Pro, Claude 3.5, and Gemini 2.0—through a systematic comparative analysis across three programming languages: Python, Java, and Swift. The evaluation framework considers multiple aspects, including overall accuracy, code efficiency, time complexity, space complexity, and multi-solution generation capabilities.The experimental results reveal substantial variations among models: OpenAI o1 Pro and Gemini achieve the highest accuracy, GPT-4o generates the most concise code, and Claude 3.5 produces the greatest number of alternative solutions. However, all models exhibit lower performance in Swift compared to Python and Java, likely due to the limited availability of training data in Swift. An in-depth error analysis identifies differences in model adaptability across programming languages and highlights key limitations of AI-assisted programming. These findings provide insights for developers and users of AI-assisted programming tools, supporting more informed decision-making in selecting and applying these technologies in different programming contexts.

Keywords: Ai-Assisted Programming, Code Generation, Large Language Models

1. Introduction

In recent years, artificial intelligence (AI) has experienced rapid development, particularly in the fields of natural language processing and machine learning. As these technologies continue to advance, an increasing number of industries have begun integrating AI into their workflows, sparking transformative changes across various sectors. Within this context, AI-assisted programming has garnered significant attention. AI systems can assist developers by interpreting error messages, identifying and analyzing bugs, and providing conceptual expansion through interactive dialogues. Moreover, AI can generate relevant code based on given requirements or a developer's conceptual framework, shifting the programmer's role from "writing code from scratch" to "reading, evaluating, and refining existing code." This shift significantly reduces the burden of coding and enhances overall efficiency [1]. Against this backdrop, the primary aim of this study is to assess the effectiveness of several leading Large Language models in programming tasks, specifically GPT-4o, OpenAI o1,

DOI: 10.54254/2755-2721/2025.TJ22242

Claude 3.5, and Gemini 2.0. By subjecting these models to a variety of code generation tasks, this research seeks to explore their capabilities and overall performance.

Since the initial release of GPT-3 by OpenAI in 2020, AI tools have undergone rapid growth and iterations. For instance, OpenAI's models have evolved from GPT-3, launched in mid-2020, to GPT-3.5 in late 2022, followed by OpenAI o1 in 2023, and the release of o1 pro in 2024. Each iteration has brought improvements in model size, optimization of pretraining datasets, and more advanced reinforcement learning methods. These advancements have significantly enhanced the accuracy of code generation, increased the ability to handle complex algorithmic tasks, and strengthened the model's reasoning capabilities across various programming languages. In addition, other public AI models have also been released. For example, Claude, developed by Anthropic, has shown outstanding performance in code generation, automated testing, and debugging across multiple programming languages. Furthermore, in 2023, Google launched Bard (now rebranded as Gemini), with continuous iterations and improvements. As these AI tools evolve, their potential for enhancing the accuracy and usability of AI-assisted programming continues to grow, alleviating concerns about potential errors and misguidance.

As AI models continue to evolve, tests assessing the programming capabilities of different AI models have also progressed. However, existing research on this topic tends to exhibit several limitations [2, 3]. For instance, many studies focus primarily on evaluating a single model [4-6], with limited comparative analysis between different models. Additionally, the programming languages used for testing are predominantly mainstream programming languages such as Python and Java, leaving a gap in the testing of less common programming languages [3]. These limitations may result in an incomplete evaluation of the AI models' programming capabilities, failing to fully capture their performance across diverse programming environments.

To address the gaps in current research, this study takes a more comprehensive approach by comparing multiple leading Large Language models (e.g., GPT-4o, OpenAI o1, Claude 3.5, and Gemini 2.0). It also includes cross-programming language comparisons, evaluating model performance across both mainstream and less common programming languages. This quantitative methodology provides a broader and more nuanced understanding of the models' capabilities, addressing the limitations of previous studies that focused on a single model or programming language. The main objectives are as follows:

- To evaluate the efficiency of various models released by OpenAI, as well as Claude 3.5 and Gemini 2.0, across a range of code generation tasks.
- To compare the performance of these Large Language models under different programming language environments.
- To identify the challenges and limitations associated with using Large Language models in programming.

2. Related Work

2.1. Code Generation

2.1.1. Code Generation Models

Before the emergence of deep learning and Large Language models, research on automated code generation primarily focused on "program synthesis" and traditional rule- or template-based approaches. Program synthesis typically relies on specific types of specifications, logical reasoning, or input–output examples to produce small-scale programs that meet user requirements. As an example, Gulwani [7] describes a method for string processing that leverages user-provided examples. This strategy not only decreases the prior knowledge required of developers but also significantly

DOI: 10.54254/2755-2721/2025.TJ22242

enhances automation in certain specialized settings. In contrast, another common approach involves the use of domain-specific languages (DSLs) or template-driven methods, where predefined syntactic rules or scaffolding enable rapid creation of project skeletons or repetitive code [8]. Although these two categories of methods do alleviate some of the burden on developers, they both lack broad generality and demand a high level of technical proficiency from users, rendering them relatively limited in terms of extensibility. Consequently, they tend to be confined to specialized programming contexts, and thus have not seen widespread adoption in more complex or diverse development needs. The following sections further examine how deep learning and Large Language models have capitalized on these foundations to achieve notable breakthroughs and expansions.

Early sequence-to-sequence (Seq2Seq) approaches, typically using RNN/LSTM encoder-decoder structures [9], showed promise yet struggled with lengthy contexts and the complex syntax–semantics of code. To mitigate these limitations, the Transformer architecture [10] introduced self-attention, improving parallel computation and handling of long-range dependencies. Consequently, Transformer-based pretrained models—such as Code2Vec [11] and CodeBERT [12] —achieved notable success in tasks including code search, clone detection, and comment generation, with CodeBERT in particular demonstrating strong multilingual capabilities [13]. Beyond these models, deep learning–based code completion tools (e.g., TabNine) embed neural networks into IDEs to provide context-aware suggestions, but still face challenges in covering large-scale project contexts, managing complex dependencies, and lowering the technical barriers to widespread "mass-market" adoption.

In recent years, the field of code generation has undergone significant transformation with the emergence of Large Language models (LLMs) such as GPT-3 and later versions (Generative Pre-trained Transformer). Leveraging the Transformer architecture's strong capability to capture long-range dependencies, these models are trained on vast corpora of text and code, thus improving performance on specific tasks while addressing the limitations of earlier Seq2Seq and traditional Transformer-based systems [14]. More importantly, GPT-based methods often adopt dialogue- or prompt-oriented interfaces, enabling users to iteratively refine or debug code through natural language, which not only lowers the technical barrier but also propels these tools toward broader public adoption [15]. Exemplified by OpenAI Codex and GitHub Copilot, GPT-driven solutions can dynamically assist with code completion, error correction, and annotation, benefiting both novice and experienced developers alike [16]. The development and iteration of generative AI technologies have seen significant progress in recent years. OpenAI has introduced its model "OpenAI o1," Google has developed the "Gemini" model, and Anthropic has advanced its "Claude" system, all contributing to the enhancement of code generation capabilities. These advancements have fostered rapid progress in dialogue-based code generation, broadening its potential applications in software development and related fields.

2.1.2. Evaluation Datasets

Initially, code generation evaluation datasets were often limited to simple algorithmic tasks in a single programming language, such as the small-scale problem sets provided by online coding platforms [17, 18]. These early datasets primarily served to validate a model's feasibility in basic algorithmic and syntactic aspects. However, as research advanced and Large Language models (LLMs) gained prominence, relying on single-programming language and relatively small collections of problems proved insufficient for comprehensively assessing a model's generality and practical applicability. Researchers thus shifted their focus to multi-programming language, medium- and large-scale datasets, exemplified by HumanEval [16] and APPS [18]. The former targets the function-generation capabilities of large models like Codex in Python, supplementing tasks with unit tests to quantify correctness, whereas the latter compiles programming competition problems ranging from basic to

DOI: 10.54254/2755-2721/2025.TJ22242

advanced, thereby more effectively evaluating a model's algorithmic proficiency and logical reasoning.

With the emergence of GPT-series models [14], whose ease of use and conversational interfaces expanded the scope of feasible evaluation strategies, researchers have increasingly adopted multi-programming language or cross-programming language datasets to investigate how well these models comprehend and transfer knowledge across diverse programming language features [19]. For instance, some platforms have begun extensively gathering questions and solutions from programming competitions or job interviews in open-source ecosystems, deploying these resources to evaluate a model's capability in multi-programming language settings.

In recent years, new evaluation benchmarks have been proposed to more comprehensively assess code generation models' capabilities. For example, BigCodeBench is a novel code generation benchmark that improves upon existing ones like HumanEval by incorporating more diverse function calls and complex instructions, aiming to evaluate the true programming capabilities of Large Language models in a more realistic setting [20]. Additionally, MEGAVERSE is a benchmark that evaluates Large Language models across natural languages, modalities, models, and tasks, with a particular focus on non-English languages. This benchmark comprises 22 datasets covering 83 natural languages, including low-resource African languages, as well as two multimodal datasets, providing a comprehensive assessment of models' multilingual and multimodal capabilities [21].

These new evaluation benchmarks offer researchers more comprehensive tools to assess Large Language models' performance in code generation tasks, particularly in multilingual and real-world application scenarios.

2.1.3. Evaluation Standards

When evaluating code generation models, different standards are typically adopted depending on the focus or specific requirements. For instance, when the primary concern is whether the model can generate correct code, metrics such as accuracy or Pass@k are often employed. An illustrative example is the HumanEval benchmark proposed by Chen, et al. [16] for assessing Codex, which quantifies code generation accuracy through a large collection of Python function tasks and corresponding test cases. Conversely, when the emphasis lies in aspects like readability and maintainability, researchers generally utilize static analysis tools or manual review procedures to evaluate code quality—such as the empirical study conducted by Nguyen and Nadi [22] on GitHub Copilot—by combining lint tools with developer feedback to systematically investigate potential security issues and maintainability metrics [18, 23]. Moreover, to address the "one problem, multiple solutions" phenomenon and account for the unique syntactic and semantic characteristics of code, researchers have proposed various evaluation methods. For instance, Zhangyin, et al. [12] introduced CodeBERT, which laid a foundation for semantic similarity metrics in code understanding; Ren, et al. [23] explored graph-based approaches to capture structural information in code; and Wang, et al. [24] highlighted the importance of alignment at both the token and structural levels. Building on these advancements, CodeBLEU was developed as a more code-specific automated evaluation metric, extending traditional text similarity measures by incorporating syntax and semantic matching, making it particularly suitable for code-related tasks [25].

2.2. Code Generation Application

2.2.1. Student Use Cases

With the rapid development of generative AI technologies, education has become a key domain where these tools are widely applied. For students, generative AI has introduced transformative changes in the way they learn and understand programming concepts.

DOI: 10.54254/2755-2721/2025.TJ22242

For instance, students can submit complex code examples to generative AI tools, requesting them to refactor the code for improved readability or to decompose intricate functions into more manageable and comprehensible segments [26]. These strategies enable students to better process programming materials and enhance their understanding of both the code itself and broader programming concepts [27]. Furthermore, students can articulate their ideas in natural language, allowing the AI to generate syntactically rigorous code based on their thought processes. This approach enables students to focus on constructing their ideas and strategies using natural language, without the need to concentrate excessively on complex code details. According to a survey conducted by Yan, et al. [1], many students found this method highly practical, as it allowed them to concentrate on higher-level programming logic without being hindered by intricate syntax. This not only improved their learning experience but also fostered critical thinking and creativity [28].

In addition, Lau and Guo [26] highlighted in their study that a significant portion of current programming curricula focuses on teaching students to memorize syntax and mechanisms. This approach often frustrates students who are more inclined toward creative projects. Generative AI, by assisting with code generation, allows students to concentrate more on creative design and problem-solving processes during class. This shift not only enhances students' learning motivation but also further stimulates their interest in programming and fosters their creativity.

2.2.2. Applications in the Workplace

Although students primarily use generative AI to support their learning process, educators and professional programmers utilize these tools to meet their unique demands. By leveraging generative AI for code generation, they can significantly enhance the efficiency of their teaching or professional workflows.

For example, in the context of programming education, example-based learning is often regarded as an effective teaching strategy [29]. A complete programming example typically includes a problem statement, a feasible solution, and detailed line-by-line explanations. However, for educators, creating a sufficient number of programming exercises along with corresponding solutions and explanations is often a time-consuming task. In practice, educators tend to provide only a limited number of examples, which may fail to fully meet students' learning needs in programming [30]. By automating the generation of these teaching resources, generative AI enables educators to maintain the same level of teaching effort while improving students' learning efficiency, ultimately enhancing the overall effectiveness of teaching [31].

For programmers, using AI to assist in code generation significantly improves productivity. For instance, a study conducted by Cui et al. (2024) at companies such as Microsoft and Accenture demonstrated that developers utilizing AI coding assistants achieved approximately 25% higher task completion rates compared to those who did not use such tools. The productivity boost was particularly significant for less experienced developers. While AI-generated code may not always be entirely accurate, its primary value lies in shifting the process from "writing code from scratch" to "reviewing and modifying AI-generated code." This shift greatly reduces the time required to complete coding tasks.

3. Methodology

This study aims to evaluate the performance of Large Language Models (LLMs) in programming tasks and compare the differences in code generation, logical reasoning, and problem-solving abilities among different models. We selected five representative LLMs (GPT-4o, OpenAI-o1, OpenAI-o1 pro, Claude 3.5, and Gemini 2.0) and used the programming problems provided by the LeetCode

DOI: 10.54254/2755-2721/2025.TJ22242

platform[1] as the test dataset. To comprehensively evaluate the performance of LLMs under different programming paradigms, we chose three programming languages: Python, Java, and Swift. The evaluation metrics include total score, average lines of code, time/space complexity, and multi-answer generation capability. Through multi-dimensional evaluation, we aim to gain an in-depth understanding of the capabilities of LLMs in programming tasks and provide a reference for future research.

3.1. Selection of AI System

This study selected five Large Language models (LLMs) for analysis, each chosen for specific reasons to provide a comprehensive evaluation of LLM capabilities:

GPT-4o: As the culmination of the ChatGPT series, GPT-4o represents the pinnacle of its natural language processing (NLP) capabilities. Its selection provides an ideal baseline for tracing the evolutionary trajectory and performance improvements within this model family [32].

OpenAI-o1: OpenAI's o1-based series is purported to employ a more advanced reasoning methodology and a tighter logical framework, demonstrating superior performance in complex problem-solving and scientific inquiry. OpenAI-o1 was therefore included to specifically investigate these advancements in logical reasoning [33].

OpenAI-o1 pro: The most recent iteration from OpenAI, OpenAI-o1 pro, builds upon OpenAI-o1 by extending processing time, theoretically leading to more reliable and in-depth responses. Despite the increased computational cost, OpenAI-o1 pro was included to explore model performance on particularly intricate and demanding tasks [34].

Claude 3.5: Developed by Anthropic, Claude 3.5 is recognized for its exceptional performance in programming tasks. Its inclusion provides a valuable benchmark against the ChatGPT series, particularly in the area of code generation [35].

Gemini 2.0: Developed by Google, Gemini 2.0 has been reported to exhibit robust performance in advanced reasoning tasks, encompassing mathematics and coding. This capability makes it well-suited for evaluating model proficiencies in logical reasoning and problem-solving [36].

This diverse selection of LLMs, spanning different architectures and specialized strengths, allows for a thorough comparison of their performance across a range of programming challenges.

3.2. Selection of Dataset

In this study, we employ coding problems from the LeetCode platform as our primary testing dataset, for the following reasons:

- **Extensive and Diverse Problem Set**
- LeetCode offers a large and systematically tiered set of problems, spanning a broad range of difficulties and covering various common algorithms and data structures. Compared to other potential problem sets, LeetCode is widely used in the industry, thus making it more representative and comparable in a research context.
- **Primarily Algorithmic Tasks with Clear Difficulty Labels**
- LeetCode's problems predominantly focus on algorithmic challenges and data structure exercises, with labels such as Easy, Medium, and Hard. These straightforward difficulty levels enable a direct and layered evaluation of how different AI models perform under varying degrees of complexity.
- **Multi-Programming Language Support**

1 LeetCode is a popular online platform for coding practice and competitive programming, providing a wide range of coding challenges. Website: https://leetcode.com

DOI: 10.54254/2755-2721/2025.TJ22242

- LeetCode's problems offer solutions in multiple programming languages, enabling cross-pro gramming language comparisons of model performance under a consistent set of tasks.
- **Convenient Online Testing Environment**
- LeetCode's built-in code execution and testing functionality allows for rapid verification of whether AI-generated solutions pass official test cases. This feature substantially reduces the testing overhead and enhances both objectivity and reproducibility of the results in research scenarios.

Overall, the combination of clear difficulty tiers, multi-programming language support, and a user-friendly online testing framework makes LeetCode an ideal platform for assessing the performance of Large Language models in programming tasks.

3.3. Selection of Programming Language

This study selects three programming languages—Python, Java, and Swift—to evaluate the performance of LLMs in different programming language environments. Python is the leading programming language in the AI field, particularly in machine learning and deep learning, with many foundational AI models being built using Python [37]. Therefore, Python is chosen as the first programming language to assess the performance of LLMs in a mainstream language within the AI domain. Although Java is less widely used in AI compared to Python, it is an overall mainstream programming language with extensive application across various domains [38]. Thus, Java is selected as the second programming language to test the performance of LLMs in another widely used language. Swift, primarily used for iOS development, has a smaller user base and is considered a relatively niche programming language. However, due to its more limited usage, Swift provides a valuable contrast and allows for the assessment of LLM performance in a low-resource language environment [38]. Therefore, Swift is selected as the third programming language for testing.

3.4. Evaluation Strategy

In this study, we employ the following evaluation metrics to comprehensively assess the performance of the models:

Total Score: The LeetCode platform provides a set of test cases for each problem. For each model-generated solution, a score of 1 is awarded for every passed test case. The total score for each model is the sum of scores obtained across all problems. This metric directly reflects the model's accuracy in solving programming problems.

Average Lines of Code: The number of lines of code serves as an indicator of the conciseness and efficiency of the generated code. Utilizing existing libraries or predefined functions effectively reduces the number of lines required. A lower average lines of code count suggests a higher level of proficiency in programming language usage and efficient coding practices.

Time/Space Complexity: LeetCode provides runtime and memory usage data for each submission. However, given the potential for fluctuations in these measurements on the platform, this study evaluates models based on their time and space complexity. Specifically, for a given problem, the time/space complexities of the solutions generated by different models are compared. For instance, if four models produce solutions with O(n) time complexity and one model produces a solution with O(n^2) time complexity, the O(n^2) solution is assigned a score of 0, while the O(n) solutions are each assigned a score of 1. This metric reflects the efficiency of the generated code.

Number of Multiple Answer Responses: In some instances, the models may generate multiple potential solutions for a given problem. This capability is considered to reflect the model's ability to analyze and approach problems from multiple perspectives, providing diverse and potentially

DOI: 10.54254/2755-2721/2025.TJ22242

valuable solutions. Therefore, the number of multiple answer responses is included as an evaluation metric to assess the model's comprehensive problem-solving capabilities.

4. Experiment Results

4.1. Overall Performance Comparison

This section presents the results of the evaluation conducted on multiple models across different programming languages. The models assessed include GPT 4o, OpenAI o1, OpenAI o1 pro, Claude 3.5, and Gemini 2.0. The performance of these models was evaluated based on several key metrics: overall score, number of correct answers, average lines of code, time complexity score, space complexity score, and the number of multiple-answer situations. As shown in Table 1, the performance of the models varies significantly across different programming languages.

Table 1 : Model Performance Comparison Across Programming Languages

Model	Porgramming Language	Overall Score	Average Lines of Code	Time Complexity Score	Space Complexity Score	Multiple-Answers Count
GPT-4o	Python3	376	13.3	9	6	5
	Java	373	20.2	4	3	3
	Swift	351	20.1	12	3	2
OpenAI-o1	Python3	376	17.1	12	6	15
	Java	377	22.3	9	3	20
	Swift	351	22.6	14	3	5
OpenAI-o1 pro	Python3	378	15.5	14	3	24
	Java	377	22.1	5	2	19
	Swift	365	23.4	18	2	20
Claude 3.5	Python3	370	14.1	10	3	34
	Java	371	20.8	10	0	15
	Swift	345	21.4	12	0	6
Gemini 2.0	Python3	378	13.7	6	3	10
	Java	381	21.3	2	3	9
	Swift	352	21.2	12	3	6

4.1.1. Overall Score

Figure 1 presents a comparative performance analysis of various models across different programming languages. As shown, the models exhibit similar performance in both Python and Java, with a score rate of 97% or higher out of the maximum possible score of 383. In contrast, the models' performance in Swift, a less commonly used programming language, shows a notable decline. From an inter-model comparison perspective, Gemini achieves the highest scores in both Python and Java, outperforming all other models. The second-best performing model is OpenAI o1 pro, which surpasses both OpenAI o1 and GPT 4o. On the other hand, Claude 3.5 demonstrates the weakest performance, scoring lower than all other models in both Python and Java. In the case of Swift, OpenAI o1 pro stands out with a significant performance gap compared to the other models. Overall, Gemini performs well in both Python and Java, but the performance gap between it and other well-performing models, such as o1 pro, is marginal. In contrast, o1 pro performs the best in Swift, with a noticeable performance gap over the other models, demonstrating a clear advantage in competence.

DOI: 10.54254/2755-2721/2025.TJ22242

This highlights the superior capability of OpenAI o1 pro in handling Swift, a relatively less widely used programming language.

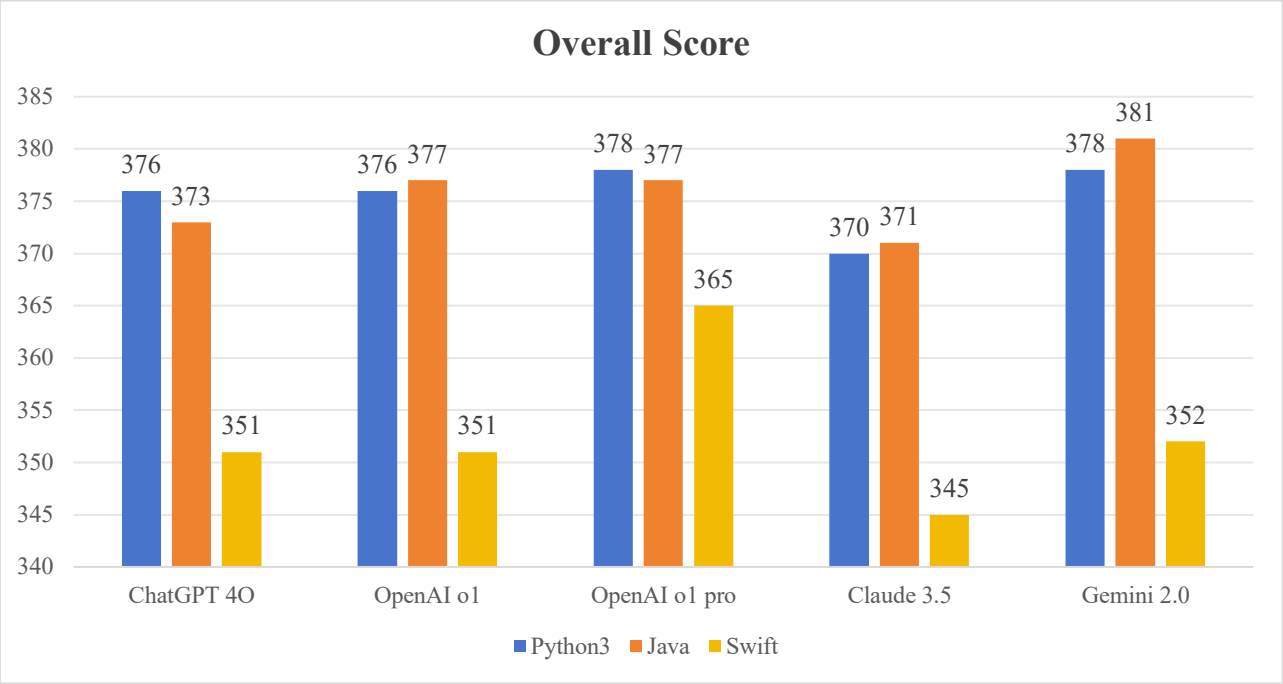

Figure 1 : Model Performance Comparison Across Programming Languages.

4.1.2. Average Lines of Code

As illustrated in Figure 2, GPT-4o generates the most concise code in both Python and Java, producing significantly fewer average lines of code compared to the other models. This effect is particularly pronounced in Python, where GPT-4o exhibits a substantially lower average line count, indicating high efficiency in code generation for these two programming languages. However, in Swift, GPT-4o produces a notably higher number of lines of code than the other models, suggesting reduced efficiency in code generation for this programming language.

In contrast, OpenAI o1, OpenAI o1 pro, Claude 3.5, and Gemini 2.0 demonstrate relatively similar performance in terms of average lines of code generated. While their line counts generally exceed those of GPT-4o, the differences are not as pronounced. Notably, Gemini 2.0, despite generating slightly more lines of code, maintains a relatively high level of accuracy, indicating an effective understanding and application of programming languages.

Overall, while GPT-4o demonstrates superior performance in Python and Java, Gemini 2.0 exhibits strong cross-programming language capabilities by producing relatively concise code while maintaining high accuracy. This suggests a deeper comprehension of programming languages and an enhanced ability to generate efficient code across multiple programming environments.

DOI: 10.54254/2755-2721/2025.TJ22242

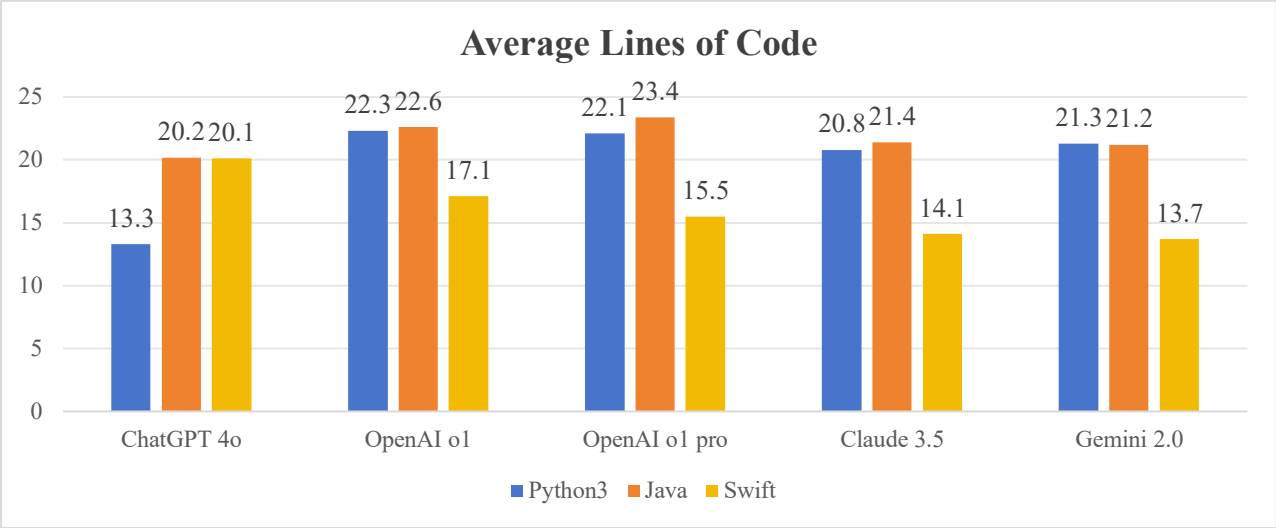

Figure 2 : Average Lines of Code Generated by Different Models.

4.1.3. Time complexity and Space complexity

From Figure 3, it can be observed that, with respect to time complexity, the models perform significantly better in Swift compared to Python and Java. According to our scoring methodology, a score of 0 for all AI models in a given language indicates that all models provide the same complexity answers for all programming problems within that language. Therefore, a higher total score for models in a particular language suggests a greater tendency for different models to produce varying complexity answers, indicating greater instability in terms of complexity in their responses. The information from the figure illustrates that, in Swift, there is considerable variation in the time complexity answers provided by different models across different problems, with this variation consistently observed across multiple problems. In contrast, Java receives the lowest scores among the three programming languages, suggesting that, when addressing programming problems in Java, the models tend to provide more consistent time complexity answers. This disparity can be attributed to the training data discussed earlier. As Swift is a less commonly used programming language, it likely lacks sufficient training samples, leading to a lack of reference templates when models encounter Swift-related problems, which results in greater variation in time complexity. Conversely, Java, being a widely used language, benefits from a larger number of training samples, allowing models to generate more consistent time complexity answers for Java-related problems based on the best reference templates, thus resulting in more stable time complexity.

Additionally, OpenAI o1 Pro achieves the highest scores in both Swift and Python, indicating that, when coding in these languages, OpenAI o1 Pro is more likely to provide the lowest time complexity answers across a greater number of problems compared to other models. This further highlights the superior performance of OpenAI o1 Pro in handling Swift, a less widely used programming language, compared to other models.

From Figure 4**,** it can be observed that the overall scores of different models in terms of space complexity are relatively low, and in all three programming languages, these scores are significantly lower than the scores for time complexity. This suggests that, when addressing programming problems, the models tend to provide consistent space complexity answers across the three programming languages. In other words, the space complexity of the answers generated by AI models remains relatively stable.

DOI: 10.54254/2755-2721/2025.TJ22242

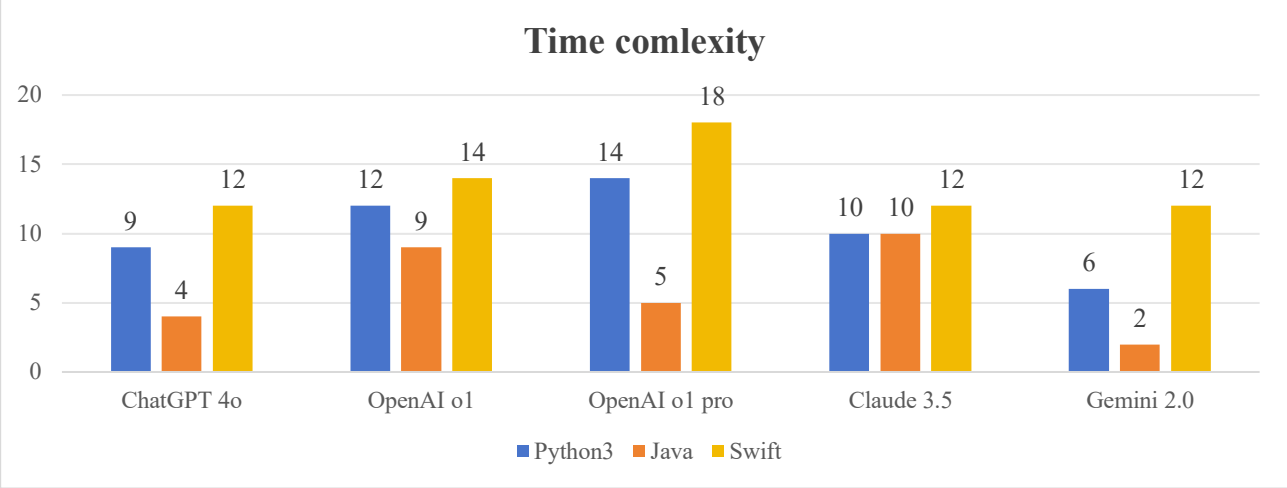

Figure 3 : Time Complexity Scores of Different Models.

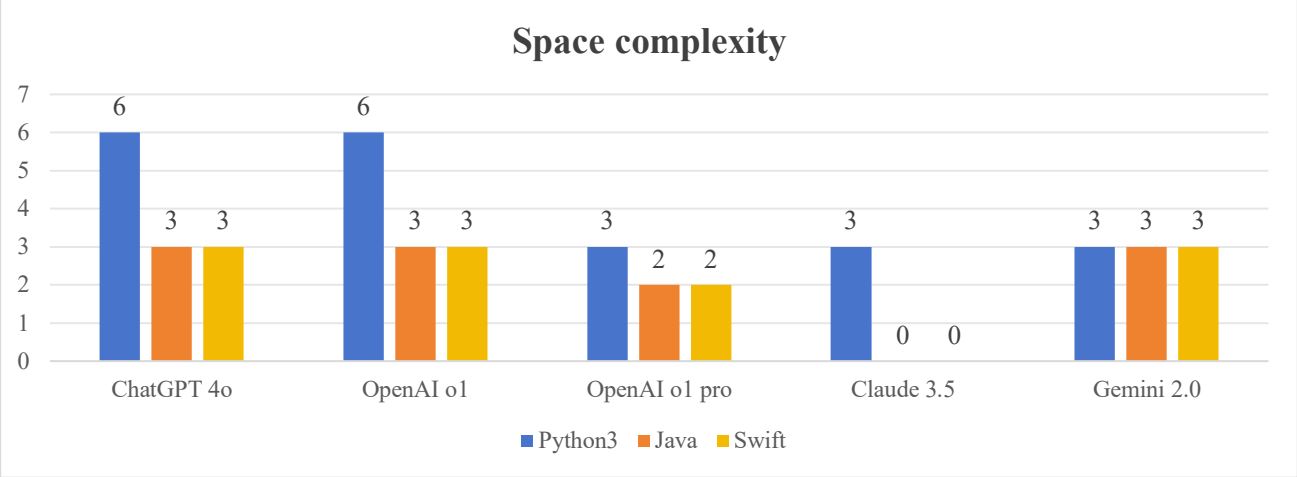

Figure 4 : Space Complexity Scores of Different Models.

4.1.4. Multiple-Answers Count

Figure 5 illustrates significant variations in the number of multiple answers generated across different models and programming languages. Among these models, Claude 3.5 stands out, particularly in Python3, followed by OpenAI o1 pro. This suggests that these models are particularly effective at generating multiple answers for Python3.

In contrast, GPT 4o and Gemini 2.0 generate relatively fewer answers across all programming languages, with GPT 4o showing the lowest performance in Swift and Java. Similarly, Gemini 2.0 also generates fewer answers in Java and Swift.

It is noteworthy that o1 Pro provides a relatively balanced number of multiple answers across all three programming languages, with a significantly higher number of answers generated in Swift compared to other models. This suggests that o1 Pro exhibits strong adaptability in generating multiple answers, offering diverse solutions that are not constrained by the programming language itself, and demonstrates superior proficiency in handling the Swift language compared to other models.

DOI: 10.54254/2755-2721/2025.TJ22242

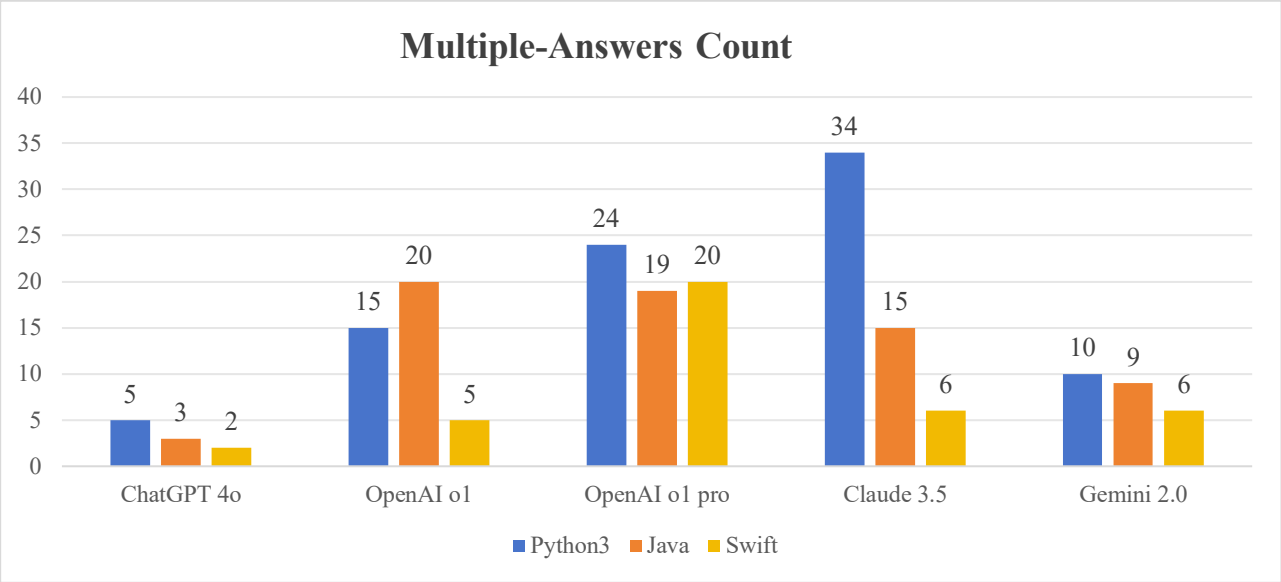

Figure 5 : Multiple-Answers Count of Different Models.

4.2. Expanded Performance Analysis

4.2.1. Performance Analysis by Task Difficulty

Based on the data presented in Table 2 and Figure 6, it can be inferred that, when the problem difficulty is relatively low, models with superior capabilities, such as OpenAI o1 pro, do not fully exhibit their potential advantages. Specifically, in the easy difficulty category, due to inherent instability in the AI model responses, OpenAI o1 pro occasionally produces errors on simpler tasks, even though such errors are not observed in the responses of GPT-4o and OpenAI o1. As a result, o1 pro ranks fourth in terms of accuracy in the easy category.

However, as the difficulty level increases—particularly in the medium and hard categories—the advantages of higher-performing models become more pronounced. These models demonstrate superior adaptability and problem-solving abilities, especially in dealing with complex issues. Notably, OpenAI o1 pro shows a progressively widening gap in its score rate compared to other models. In the easy category, it ranks fourth, but in the medium category, it rises to first place. In the hard category, although all models experience a decline in their scores, o1 pro's performance decreases at a slower rate and it retains its leading position. This suggests that, as the complexity of the problems increases, the true strengths of these models are realized, particularly for those that initially showed weaker performance on simpler tasks but performed better on more complex problems.

In contrast, Claude 3.5, while ranking lowest in the overall score, shows a noticeable decline in performance as the difficulty increases. Nevertheless, in the easy category, it maintains a stable performance, scoring just below Gemini 2.0, and outperforming all the OpenAI models. This indicates that while Claude 3.5 struggles with more complex problems, it performs relatively well on simpler tasks.

DOI: 10.54254/2755-2721/2025.TJ22242

Table 2 : The Specific Scores of Different Models across Different Difficulty Levels.

Model	Programming Language	Easy	Medium	Hard
GPT-4o	Python3	132	119	125
	Java	131	119	123
	Swift	126	106	119
OpenAI-o1	Python3	132	119	125
	Java	129	119	129
	Swift	125	105	121
OpenAI-o1 pro	Python3	130	119	129
	Java	132	119	126
	Swift	125	111	129
Claude 3.5	Python3	132	119	119
	Java	133	115	123
	Swift	126	110	109
Gemini 2.0	Python3	132	119	127
	Java	132	119	130
	Swift	129	107	116

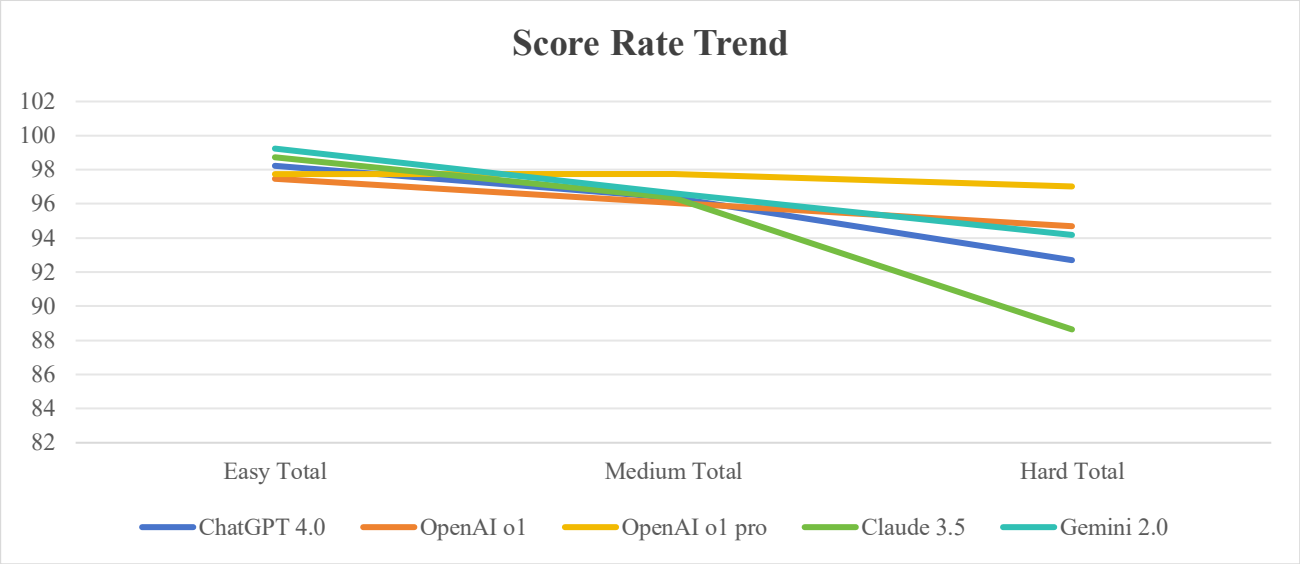

Figure 6 : Trend of Model Scoring Rates.

4.2.2. Thinking Time Analysis

In this study, we specifically recorded the average thinking times of the OpenAI o1 and OpenAI o1 pro models when processing problems of varying difficulty levels. As shown in Table 3, the detailed data illustrate the distribution of thinking times across different difficulty levels for these models. Although the thinking times of other models are not displayed, they generally exhibit response times close to zero seconds, indicating minimal delay in generating answers.

This disparity aligns with psychologist Daniel Kahneman's "Thinking, Fast and Slow" theory [39], which distinguishes between "System 1" (fast, intuitive thinking) and "System 2" (slow, deliberate thinking). In this framework, System 1 is characterized by quick, automatic responses to simple tasks, while System 2 involves thoughtful, logical reasoning when faced with complex problems.

DOI: 10.54254/2755-2721/2025.TJ22242

Recent advancements have applied this theory to AI model design. For instance, Google's DeepMind researchers developed the Talker-Reasoner framework, enabling AI agents to perform both rapid and deliberate thinking to better handle complex tasks [40].

OpenAI's official documentation indicates that the thinking mechanism of OpenAI o1 and its derivatives aligns with the characteristics of System 2 [33], exhibiting a typical Think Slow approach. Before generating responses, these models engage in deeper reasoning and computation, whereas models with shorter response times are more likely to rely on a Think Fast strategy for rapid decision-making.

Notably, even among the o1 and o1 pro models, the average thinking time of o1 pro is approximately ten times longer than that of o1. Considering our previously recorded accuracy rates and overall scores, the increased thinking time did not significantly enhance accuracy for lower-difficulty questions (classified as Easy and Medium in this study). However, for higher-difficulty questions (classified as Hard), the extended thinking time resulted in a notable improvement in efficiency.

Nevertheless, whether such prolonged thinking times justify the modest accuracy gains remains a topic for further discussion. In practical applications, balancing thinking time with response speed to meet the demands of various scenarios is an area warranting deeper exploration.

Table 3 : Comparison of Thinking Time.

Model	Programming Language	Average Thinking Time (s)
OpenAI-o1	Python3	4
	Java	6.5
	Swift	8.0
OpenAI-o1 pro	Python3	75.8
	Java	80.3
	Swift	86.4

4.2.3. Analysis of Incorrect Answer Types

In this study, we categorized the errors made by different models in their responses into four types: argument or operation errors, runtime errors, answer discrepancy errors, and output format errors.

Argument or operation error refer to fundamental mistakes that occur when using functions incorrectly or performing invalid operations on data structures. For instance, in the Swift code written by Claude 3.5 for the Wordladder II problem, the code attempts to access a property of an optional type without first unwrapping it. In Swift, optional types must be unwrapped before accessing their properties. The failure to perform this unwrapping step led to a compile error, causing the entire code to be non-functional. Such errors are indicative of a lack of proficiency with the programming language, often occurring among novice programmers.

Runtime error occur when there are no syntax errors or issues with data structure operations, but problems arise during array manipulation due to insufficient consideration of boundary conditions. For example, attempting to access an element outside the bounds of an array can result in a runtime error. This error type indicates a failure to account for special cases in the problem-solving logic. While more advanced than simple syntax errors, runtime errors still cause program crashes due to accessing non-existent array elements, highlighting the lack of robustness in the code.

Answer discrepancy error occur when the code runs correctly, but produces incorrect results for specific test cases. These errors are typically caused by insufficient consideration of edge cases and extreme values, similar to runtime errors. However, unlike runtime errors, these errors do not cause

DOI: 10.54254/2755-2721/2025.TJ22242

the program to crash, allowing the code to continue running. This makes the code more robust than in cases of runtime errors. Nevertheless, the failure to account for certain conditions still leads to incorrect outputs in specific scenarios.

Output format error occur when the output differs from the expected result, not due to content discrepancies, but due to formatting issues, such as extra spaces or punctuation differences. While these errors are flagged as incorrect by the automated system, manual verification by the researchers confirms that the content matches the expected result. Although automated systems classify these as errors, they are generally considered more acceptable due to their minor nature, typically stemming from formatting issues. Even experienced programmers may encounter such errors, which are generally regarded as less critical compared to other types of errors.

Table 4 presents the error type statistics for different models across various programming languages. From the table, it is evident that the majority of errors in Python and Java are Answer Discrepancy Errors, where the code produces incorrect results for certain boundary conditions without affecting the overall functionality of the program. In contrast, Argument or Operation Errors, which stem from a lack of understanding of the programming language, and Runtime Errors, which have a more significant impact on the program's stability, are more commonly observed in Swift. This trend is consistent across all models. Therefore, we can infer that, as a less commonly used programming language, AI models may have insufficient training data for Swift, leading to a higher frequency of basic errors. Consequently, caution should be exercised when using AI to assist in writing code for rare programming languages.

Table 4 : Error Type Statistics

Model	Programming Language	Argument or operation errors	Runtime errors	Answer discrepancy errors	Output format errors
GPT-4o	Python3	0	1	4	0
	Java	0	0	5	1
	Swift	2	6	10	0
OpenAI-o1	Python3	0	0	4	0
	Java	0	0	4	0
	Swift	2	7	9	0
OpenAI-o1 pro	Python3	0	0	3	0
	Java	0	1	3	0
	Swift	0	5	5	0
Claude 3.5	Python3	0	0	7	0
	Java	1	0	6	0
	Swift	4	4	15	0
Gemini 2.0	Python3	0	0	2	0
	Java	0	0	2	0
	Swift	2	6	8	0

5. Conclusion

This study systematically evaluates the performance of several mainstream Large Language models (GPT-4o, OpenAI o1, OpenAI o1 pro, Claude 3.5, and Gemini 2.0) across a series of code generation tasks in three programming languages: Python, Java, and Swift. By analyzing key metrics such as total score, average lines of code, time complexity, space complexity, and multi-answer generation

DOI: 10.54254/2755-2721/2025.TJ22242

capability, this study provides a comprehensive comparison of the models' effectiveness in programming-related tasks.

The results indicate that OpenAI o1 pro and Gemini outperform other models in overall accuracy. Meanwhile, “Think Slow” models, represented by OpenAI o1 pro and OpenAI o1, exhibit lower accuracy in handling low-difficulty programming tasks, sometimes falling behind “Think Fast” models. However, as task complexity increases, these “Think Slow” models demonstrate greater adaptability and problem-solving capabilities, suggesting that users should balance response time and accuracy based on task complexity. On the other hand, “Think Fast” models, despite their lower accuracy, exhibit notable strengths: GPT-4o generates the most concise code, indicating efficiency in solution formulation; Claude 3.5 generates multiple possible solutions, offering users a diverse range of options, and performs well on low-difficulty problems. Gemini, although categorized as a “Think Fast” model, achieves accuracy comparable to OpenAI o1 pro and OpenAI o1, highlighting its robust overall performance.

A cross-programming language comparison indicates that all AI models perform well when handling widely used programming languages such as Python and Java. Although Gemini outperforms other models in these two languages, the performance gap between models remains relatively small. However, when processing Swift, a less commonly used programming language, the performance of all AI models declines. This decline is reflected in a reduction in accuracy and a decrease in multi-answer generation capability. Notably, OpenAI o1 Pro exhibits a more gradual decline in performance compared to other models, maintaining a substantial lead, which suggests that it is more effective in handling tasks related to Swift.

Furthermore, error type analysis provides deeper insights into model performance across different programming languages. For Java and Python tasks, most errors involve formatting inconsistencies or logical mistakes that do not affect overall program execution. However, for Swift-related tasks, models frequently encounter fundamental syntax errors, operational mistakes, and critical logical errors that can cause program crashes. This finding suggests that models exhibit lower proficiency in niche programming languages, largely due to insufficient training data. It also highlights a key limitation of LLMs in programming—performance degradation in low-resource programming languages. Therefore, developers using AI models for niche programming language programming should exercise caution, verifying AI-generated code to enhance reliability.

References

[1] *W. Yan, T. Nakajima, and R. Sawada, "Benefits and Challenges of Collaboration between Students and Conversational Generative Artificial Intelligence in Programming Learning: An Empirical Case Study," Education Sciences, vol. 14, no. 4, p. 433, 2024. [Online]. Available: https://www.mdpi.com/2227-7102/14/4/433.*

[2] *D. Z. Guo, Qihao; Yang, Dejian; Xie, Zhenda; Dong, Kai; Zhang, Wentao; Chen, Guanting; Bi, Xiao; Wu, Y.; Li, Y.K.; Luo, Fuli; Xiong, Yingfei; Liang, Wenfeng, "DeepSeek-Coder: When the Large Language Model Meets Programming -- The Rise of Code Intelligence," doi: 10.48550/arXiv.2401.14196.*

[3] *M. Izadi, J. Katzy, T. V. Dam, M. Otten, R. M. Popescu, and A. V. Deursen, "Language Models for Code Completion: A Practical Evaluation," presented at the Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, Lisbon, Portugal, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639138.*

[4] *A. Vadaparty et al., "CS1-LLM: Integrating LLMs into CS1 Instruction," presented at the Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1, Milan, Italy, 2024. [Online]. Available: https://doi.org/10.1145/3649217.3653584.*

[5] *J. Savelka, A. Agarwal, C. Bogart, Y. Song, and M. Sakr, "Can Generative Pre-trained Transformers (GPT) Pass Assessments in Higher Education Programming Courses?," presented at the Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1, Turku, Finland, 2023. [Online]. Available: https://doi.org/10.1145/3587102.3588792.*

[6] *M. Richards, K. Waugh, M. Slaymaker, M. Petre, J. Woodthorpe, and D. Gooch, "Bob or Bot: Exploring ChatGPT's Answers to University Computer Science Assessment," ACM Trans. Comput. Educ., vol. 24, no. 1, p. Article 5, 2024, doi: 10.1145/3633287.*

DOI: 10.54254/2755-2721/2025.TJ22242

[7] S. Gulwani, "Automating string processing in spreadsheets using input-output examples," presented at the Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Austin, Texas, USA, 2011. [Online]. Available: https://doi.org/10.1145/1926385.1926423.
[8] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," ACM Comput. Surv., vol. 37, no. 4, pp. 316–344, 2005, doi: 10.1145/1118890.1118892.
[9] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," presented at the Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2, Montreal, Canada, 2014.
[10] A. Vaswani et al., "Attention is all you need," presented at the Proceedings of the 31st International Conference on Neural Information Processing Systems, Long Beach, California, USA, 2017.
[11] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: learning distributed representations of code," Proc. ACM Program. Lang., vol. 3, no. POPL, p. Article 40, 2019, doi: 10.1145/3290353.
[12] F. Zhangyin et al., "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," in Findings of the Association for Computational Linguistics: EMNLP 2020, T. Cohn, Y. He, and Y. Liu, Eds., November 2020 2020: Association for Computational Linguistics, pp. 1536-1547, doi: 10.18653/v1/2020.findings-emnlp.139.
[13] Z. Feng et al., "Codebert: A pre-trained model for programming and natural languages," in Findings of the Association for Computational Linguistics: EMNLP 2020, 2020: Association for Computational Linguistics, doi: 10.18653/v1/2020.findings-emnlp.139.
[14] T. B. Brown et al., "Language models are few-shot learners," presented at the Proceedings of the 34th International Conference on Neural Information Processing Systems, Vancouver, BC, Canada, 2020.
[15] OpenAI. "ChatGPT: Optimizing language models for dialogue." OpenAI. https://openai.com/blog/chatgpt .
[16] M. Chen et al., "Evaluating large language models trained on code," arXiv preprint arXiv:2107.03374, 2021, doi: 10.48550/arXiv.2107.03374.
[17] J. Austin et al., "Program synthesis with large language models," arXiv preprint arXiv:2108.07732, 2021, doi: 10.48550/arXiv.2108.07732.
[18] D. Hendrycks et al., "Measuring coding challenge competence with apps," arXiv preprint arXiv:2105.09938, 2021, doi: 10.48550/arXiv.2105.09938.
[19] Y. Li et al., "Competition-level code generation with alphacode," Science, vol. 378, no. 6624, pp. 1092-1097, 2022, doi: 10.1126/science.abq1158.
[20] T. Y. Zhuo et al., "Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions," arXiv preprint arXiv:2406.15877, 2024, doi: 10.48550/arXiv.2406.15877.
[21] A. Sanchit et al., "MEGAVERSE: Benchmarking Large Language Models Across Languages, Modalities, Models and Tasks," presented at the Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers), 2024.
[22] N. Nguyen and S. Nadi, "An Empirical Evaluation of GitHub Copilot's Code Suggestions," presented at the 2022 Mining Software Repositories Conference, MSR 2022, 2022.
[23] S. Ren et al., CodeBLEU: a Method for Automatic Evaluation of Code Synthesis. 2020.
[24] X. Wang et al., "MAVEN: A Massive General Domain Event Detection Dataset," 2020: Association for Computational Linguistics, pp. 1652–1671, doi: 10.18653/v1/2020.emnlp-main.129.
[25] M. Chen, & Liu, L., "Codebleu: a method for automatic evaluation of code synthesis," arXiv preprint arXiv:2009.10297, 2020, doi: 10.48550/arXiv.2009.10297.
[26] S. Lau and P. Guo, "From "Ban It Till We Understand It" to "Resistance is Futile": How University Programming Instructors Plan to Adapt as More Students Use AI Code Generation and Explanation Tools such as ChatGPT and GitHub Copilot," presented at the Proceedings of the 2023 ACM Conference on International Computing Education Research - Volume 1, Chicago, IL, USA, 2023. [Online]. Available: https://doi.org/10.1145/3568813.3600138.
[27] C. E. Smith et al., "Early Adoption of Generative Artificial Intelligence in Computing Education: Emergent Student Use Cases and Perspectives in 2023," presented at the Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1, Milan, Italy, 2024. [Online]. Available: https://doi.org/10.1145/3649217.3653575.
[28] F. Mosaiyebzadeh, S. Pouriyeh, R. Parizi, N. Dehbozorgi, M. Dorodchi, and D. M. Batista, "Exploring the Role of ChatGPT in Education: Applications and Challenges," presented at the Proceedings of the 24th Annual Conference on Information Technology Education, Marietta, GA, USA, 2023. [Online]. Available: https://doi.org/10.1145/3585059.3611445.
[29] S.-S. Abdul-Rahman and B. du Boulay, "Learning programming via worked-examples: Relation of learning styles to cognitive load," Computers in Human Behavior, vol. 30, pp. 286-298, 2014/01/01/ 2014, doi: https://doi.org/10.1016/j.chb.2013.09.007.
[30] B. Jury, A. Lorusso, J. Leinonen, P. Denny, and A. Luxton-Reilly, "Evaluating LLM-generated Worked Examples in an Introductory Programming Course," presented at the Proceedings of the 26th Australasian Compu

DOI: 10.54254/2755-2721/2025.TJ22242

ting Education Conference, Sydney, NSW, Australia, 2024. [Online]. Available: https://doi.org/10.1145/3636243.3636252.

[31] J. Leinonen et al., "Comparing Code Explanations Created by Students and Large Language Models," presented at the Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1, Turku, Finland, 2023. [Online]. Available: https://doi.org/10.1145/3587102.3588785.

[32] OpenAI. "Hello GPT-4o." OpenAI. https://openai.com/index/hello-gpt-4o/ .

[33] OpenAI. "Introducing OpenAI o1-preview." OpenAI. https://openai.com/index/introducing-openai-o1-preview/ .

[34] OpenAI. "Introducing ChatGPT Pro." OpenAI. https://openai.com/index/introducing-chatgpt-pro/ .

[35] Anthropic. "Raising the bar on SWE-bench Verified with Claude 3.5 Sonnet." Anthropic. https://www.anthropic.com/research/swe-bench-sonnet .

[36] G. DeepMind. "Introducing Gemini 2.0: our new AI model for the agentic era." Google DeepMind. https://blog.google/technology/google-deepmind/google-gemini-ai-update-december-2024/?utm_source=deepmind.google&utm_medium=referral&utm_campaign=gdm&utm_content=#ceo-message .

[37] A. Géron, Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd ed. O'Reilly Media, 2019.

[38] S. Overflow. "Stack Overflow Developer Survey 2024." https://survey.stackoverflow.com/ .

[39] D. Kahneman, Thinking, Fast and Slow. New York: Farrar, Straus and Giroux, 2011.

[40] B. Dickson. "DeepMind's Talker-Reasoner framework brings System 2 thinking to AI agents." https://venturebeat.com/ai/deepminds-talker-reasoner-framework-brings-system-2-thinking-to-ai-agents/ .