Evaluating Generative AI Models for Code Generation Tasks Using Embedding-Based Semantic Similarity

Dominik Palla and Ondrej Krejcar

Abstract— **Generative artificial intelligence (AI) is rapidly transforming software development, especially in code generation. Large Language Models (LLMs) show strong potential for automating programming tasks, though their performance varies with task complexity. This study systematically evaluates state-of-the-art models, including OpenAI GPT-4.5 Preview, GPT-4o, GPT-4o Mini, GPT-4 Turbo, GPT-3.5 Turbo, GPT-o1, GPT-o3 Mini, Google's Gemini 1.5 Pro, Gemini 1.5 Flash, Gemini 2.0 Flash, Gemini 2.0 Flash Lite, Anthropic's Claude 3 Opus, Claude 3 Sonnet, Claude 3 Haiku, Claude 3.5 Sonnet, Claude 3.5 Haiku, Claude 3.7 Sonnet, and Meta's LLaMA 3.0 8B Instruct, LLaMA 3.1 8B Instruct. These models were tested on ten Python programming tasks—five simple and five complex—and evaluated using an embedding-based semantic similarity approach. High-performing models such as GPT-4.5 Preview and GPT-4o Mini consistently produced accurate outputs, while LLaMA 3.1 8B Instruct performed weakest. Interestingly, complex tasks yielded higher similarity scores, likely due to their structured outputs. The results highlight the need for complementary metrics beyond semantic similarity, including execution correctness and efficiency. This study offers practical insights into AI-assisted coding and points toward future research directions for improving generative models in real-world applications.**

I. Introduction

The rapid rise of generative AI has notably impacted software development, with large language models (LLMs) excelling in natural language understanding, reasoning, and code generation. AI-powered tools assist programmers by automating routine tasks, boosting productivity, and supporting learning.

Yet, the quality of AI-generated code varies—some models produce valid outputs, others generate inefficient or incorrect implementations. Reliable evaluation methods are thus essential to assess model accuracy and performance.

This study offers a comparative analysis of leading generative AI models in Python code generation. Ten programming tasks of increasing complexity were used to evaluate outputs against reference solutions from university lecturers. An embedding-based approach (OpenAI's text-embedding-3-small) was applied to compute cosine similarity and assess semantic alignment.

We propose a systematic framework for automated evaluation of generative models and analyze the performance of OpenAI GPT, Anthropic Claude, Google Gemini, and Meta LLaMA models. The findings highlight each model's strengths and weaknesses, offering insights into their practical utility for real-world programming scenarios.

II. Theoretical Framework

The recent growing use of generative artificial intelligence (AI) in software engineering has introduced new paradigms for code development, automation, and productivity. Large Language Models (LLMs), such as GPT, Claude, Gemini, and LLaMA, have shown remarkable capabilities in understanding natural language and generating program code across various programming languages and domains. This section reviews the theoretical foundations of code generation using LLMs and the challenges and methods related to their evaluation, with a specific focus on embedding-based similarity.

A. Generative AI in Software Development

The adoption of AI in software engineering has led to significant transformations in coding workflows. AI tools can assist with a variety of tasks, including feature implementation, test case generation, bug fixing, and documentation. These tools are increasingly embedded into development environments, offering real-time support to developers and potentially reshaping their roles and required skills. Aarti [1] offers an overview of modern AI-powered coding tools and concludes that these systems can streamline development by reducing manual effort while raising new concerns about code reliability and oversight. Dakhel et al. [2] extend this perspective by presenting a family of studies on code generation, underlining the opportunities and limitations of AI in automating key development activities.

France [3] situates this transformation within the broader software industry by analyzing the influence of tools such as ChatGPT and GitHub Copilot on developers' work and long-term employment prospects. The study also introduces a capability maturity model (CMM) for integrating LLMs into development workflows. Similar findings are echoed by Sajja et al. [4], who emphasize how AI affects code quality, maintainability, and developer productivity. They argue that the combination of automation and intelligent suggestions can significantly accelerate development, although ethical and security concerns remain.

D. Palla is with the University of Hradec Kralove, Faculty of informatics and management, Rokitanskeho 62, 500 03 Hradec Kralove, Czech Republic (phone: +420 730 644 187, email: dominik.palla@uhk.cz).

O. Krejcar is with the Skoda Auto University, Na Karmeli 1457, 293 01 Mlada Boleslav, Czech Republic (email: ondrej.krejcar@savs.cz).

As generative AI capabilities expand, the future of software development is likely to evolve through a range of trajectories. Sauvola et al. [5] propose multiple future scenarios that integrate automation, regulation, and ethical considerations, suggesting that developers and organizations must proactively prepare for both the opportunities and the risks of AI-based workflows.

B. *Performance and Quality of AI-Generated Code*

The quality of AI-generated code is a central concern in evaluating the practical utility of generative models. Idrisov and Schlippe [6] conducted a comparative study of AI- and human-generated code in multiple languages, using complexity metrics and execution performance. Their findings highlight considerable variability among models in terms of correctness, efficiency, and maintainability. Notably, even incorrect AI-generated solutions often required only minimal modifications to become functional, suggesting their usefulness in speeding up development cycles. Tosi [7] similarly conducted an empirical evaluation of AI-generated Java code and concluded that, although models can produce syntactically correct solutions, human supervision remains essential to ensure correctness and quality.

In more specialized tasks, such as code for human-robot interaction, Sobo et al. [8] found that performance varied dramatically across models. Claude 3.5 Sonnet significantly outperformed Gemini 1.5 Pro and ChatGPT 3.5, indicating that model choice plays a critical role in domain-specific applications.

C. *Evaluation Challenges and Benchmark Limitations*

Evaluating AI-generated code remains a nontrivial task. Traditional metrics like BLEU have been criticized for their low alignment with human judgment, particularly in code generation where semantic correctness is more important than lexical overlap. Wang et al. [9] argue that LLM-based evaluators, referred to as "LLM-as-a-judge," offer a promising alternative by providing scores that better reflect human assessments without requiring extensive test suites or reference answers.

Yadav et al. [10] further critique existing benchmarks such as Human Evaluation and MBPP, noting their over-representation of simple programming concepts and insufficient task diversity. To address this gap, they propose PythonSaga, a new benchmark with 185 prompts balanced across 38 concepts and difficulty levels. This reinforces the need for tailored evaluation datasets, as used in this study, to provide a realistic assessment of model capabilities across varying complexity.

Furthermore, the challenge of evaluating code without ground-truth test cases has led researchers to explore the use of LLMs not only for generating code, but also for creating relevant test data. Baudry et al. [11] introduced methods to automatically generate culturally and domain-appropriate test data using LLMs, highlighting a novel but underexplored direction in code evaluation pipelines.

D. *Embedding-Based Semantic Similarity*

Given the limitations of traditional lexical metrics and the overhead of functional testing, semantic similarity based on vector embeddings has emerged as a practical evaluation method. Embedding models such as OpenAI's text-embedding-3-small encode the meaning of code snippets into high-dimensional vectors, enabling similarity comparisons via cosine distance. Wang, Kuo, and Li [12] emphasize the benefits of semantic similarity approaches but also highlight risks of overfitting to standard similarity benchmarks, which may obscure performance on real-world tasks.

Mahmud et al. [13] demonstrate how similarity measures can be applied in an ensemble setting to improve code selection. Their approach aggregates syntactic, semantic, and behavioral equivalence signals to select the best candidate from multiple LLM outputs. This structured voting mechanism significantly outperforms standalone LLMs, suggesting that similarity metrics can serve both evaluative and operational purposes in code generation pipelines.

E. *Benchmarking and Domain-Specific Adaptation*

The use of standardized evaluation procedures is essential for comparing generative models. Rollman et al. [14] emphasize that even small open-source LLMs, when fine-tuned on domain-specific data, can match or outperform general-purpose models such as GPT-4o in specialized tasks like medical billing. Their benchmarking strategy accounts for accuracy, output validity, and fidelity to target formats—criteria equally important in assessing code generation outputs.

Rani et al. [15] and Ramalakshmi and Asha [16] provide additional context on the broader use cases and challenges of AI across domains, including trust, bias, overfitting, and model interpretability. Their work supports the argument that robust evaluation requires a combination of general-purpose metrics, domain-aware testing, and human-in-the-loop validation.

III. Methodology

To evaluate the accuracy and performance of generative AI models in Python code-generation tasks, we designed a structured experiment based on a carefully selected set of ten programming tasks of increasing complexity. These tasks were specifically chosen to reflect typical programming problems that students encounter during introductory and advanced programming courses at our university.

The reference solutions for each task were created by expert lecturers from the Faculty of Informatics and Management, University of Hradec Králové. These solutions represent best practices in terms of efficiency, readability, and clarity, aligning closely with the pedagogical methods used in our curriculum.

The generative AI models evaluated include recent state-of-the-art offerings from OpenAI, Anthropic, Google Gemini, and Meta LLaMA accessed via Hugging Face API. Each AI model was provided with identical prompts formulated in a standardized manner. The models' generated outputs were compared against the university-crafted reference solutions using embedding-based semantic similarity. Specifically, each solution was transformed into vector embeddings using OpenAI's text-embedding-3-small model, chosen for its balance of efficiency and representational quality. We then computed the cosine similarity between the embedding vectors of the AI-generated solutions and the reference solutions to quantify semantic alignment and accuracy.

By employing this methodology, we ensured a rigorous, replicable, and pedagogically grounded assessment of the generative AI models' code-generation capabilities.

Additionally, to ground our methodology in current academic discourse, a brief systematic literature review (SLR) focusing on Python code generation and embedding-based similarity metrics was conducted and summarized in the theoretical framework section.

A. Tested generative AI Model

The following state-of-the-art generative AI models were evaluated in this study:

- **OpenAI GPT models:** gpt-3.5-turbo, gpt-4-turbo, gpt-4o-mini, gpt-4o, gpt-o1, gpt-o3-mini, gpt-4.5-preview
- **Anthropic Claude models:** Claude 3 Sonnet, Claude 3 Haiku, Claude 3 Opus, Claude 3.5 Haiku, Claude 3.5 Sonnet, Claude 3.7 Sonnet
- **Google Gemini models:** Gemini 1.5 Flash, Gemini 1.5 Pro, Gemini 2.0 Flash, Gemini 2.0 Flash Lite
- **Meta LLaMA models:** LLaMA 3.0 8B Instruct, LLaMA 3.1 8B Instruct (HuggingFace API)

These models were chosen based on their widespread use, availability through API access, and their ability to generate Python code.

IV. Experimental setup

This chapter contains a detailed description of the programming tasks, the prompts used to instruct generative AI models, and the reference solutions crafted by university lecturers. These tasks progressively increase in complexity to ensure a robust evaluation of AI-generated Python code.

To ensure a consistent and scalable evaluation process, all selected programming tasks will be automatically executed using a Python script. The script sequentially queries each AI model with the predefined task prompts, collects the generated responses, and stores the outputs in a structured format. Afterward, the AI-generated code is compared against the reference solutions using an embedding-based similarity approach. Specifically, OpenAI's text-embedding-3-small model is used to convert the generated and reference code into vector representations, and cosine similarity is computed to assess semantic alignment. This automation allows for efficient, reproducible, and unbiased comparison across all tested AI models. The generated outputs will be systematically stored, and the evaluation script will automatically compare them against the reference solutions using cosine similarity. The results will then be exported for further statistical analysis.

1) Task 1: Prime Number Checker

Prompt: Write a Python function that checks if a given integer is a prime number. The function should return True or False.

Reference Solution:

```
def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    return True
```

2) Task 2: Merge and Sort Two Lists

Prompt: Write a Python function that takes two sorted lists of integers and returns a single merged and sorted list.

Reference Solution:

```
def merge_sorted_lists(list1, list2):
    return sorted(list1 + list2)
```

3) Task 3: Password Validation (Regex)

Prompt: Write a Python function that validates a password. It must contain at least 8 characters, including one uppercase letter, one lowercase letter, one digit, and one special character. Return True if valid, False otherwise.

Reference Solution:

```
import re

def validate_password(password):
    pattern = r'^(?=.*[a-z])(?=.*[A-
Z])(?=.*\d)(?=.*[\W_]).{8,}$'
    return bool(re.match(pattern, password))
```

4) Task 4: Word Frequency Counter

Prompt: Write a Python function that accepts a string of text and returns a dictionary with each unique word as keys and their frequencies as values.

Reference Solution:

```
from collections import Counter

def word_frequency(text):
    words = text.lower().split()
    return dict(Counter(words))
```

5) Task 5: CSV Data Processing (Average Calculator)

Prompt: Write a Python function that reads a CSV file containing student names and their scores in three subjects (Math, English, Science), and returns a dictionary with student names as keys and their average scores as values.

Reference Solution:

```
import csv

def calculate_averages(csv_filename):
    averages = {}
    with open(csv_filename, 'r') as file:
        reader = csv.DictReader(file)
        for row in reader:
            scores = [int(row['Math']), int(row['English']), 
int(row['Science'])]
            averages[row['Name']] = sum(scores) / len(scores)
    return averages
```

6) *Task 6: JSON Data Extraction*

Prompt: Write a Python function that reads JSON data representing employees (containing fields: name, department, salary) and returns a list of employee names whose salary is above a given threshold.
Reference Solution:

```
import json

def high_salary_employees(json_filename, threshold):
    with open(json_filename, 'r') as file:
        data = json.load(file)
    return [emp['name'] for emp in data if emp['salary'] > 
threshold]
```

7) *Task 7: Simple Data Visualization (Matplotlib)*

Prompt: Write a Python function that takes a dictionary of categories and numerical values and generates a bar chart saved as a PNG image file.
Reference Solution:

```
import matplotlib.pyplot as plt

def create_bar_chart(data, filename):
    categories = list(data.keys())
    values = list(data.values())
    plt.bar(categories, values)
    plt.savefig(filename)
    plt.close()
```

8) *Task 8: Web Scraping (BeautifulSoup)*

Prompt: Write a Python function that takes a URL, scrapes all hyperlinks from the page, and returns them in a list.
Reference Solution:

```
import requests
from bs4 import BeautifulSoup

def scrape_links(url):
    response = requests.get(url)
    soup = BeautifulSoup(response.text, 'html.parser')
    return [a['href'] for a in soup.find_all('a', href=True)]
```

9) *Task 9: Text Sentiment Analysis (NLTK)*

Prompt: Write a Python function that analyzes a given text and returns sentiment as either positive, neutral, or negative using NLTK's VADER sentiment analyzer.
Reference Solution:

```
from nltk.sentiment import SentimentIntensityAnalyzer

def analyze_sentiment(text):
    sia = SentimentIntensityAnalyzer()
    score = sia.polarity_scores(text)['compound']
    if score > 0.05:
        return 'positive'
    elif score < -0.05:
        return 'negative'
    else:
        return 'neutral'
```

10) *Task 10: Simple Bank Class Implementation (OOP)*

Prompt: Write a Python class named 'Bank' that allows account creation with an initial balance, deposits, withdrawals, and balance inquiry. Include appropriate error handling (e.g., insufficient funds).
Reference Solution:

```
class Bank:
    def __init__(self, initial_balance=0):
        self.balance = initial_balance

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        if amount > self.balance:
            raise ValueError("Insufficient funds")
        self.balance -= amount

    def get_balance(self):
        return self.balance
```

V. Results

This section presents the results of our evaluation of generative AI models for Python code generation. The results are analyzed based on their cosine similarity scores with the reference solutions. The section includes a ranking of models by their overall accuracy, a comparison between simple and complex tasks, and graphical representations of the performance trends.

A. *Model Performance Ranking*

The ranking of models based on their average cosine similarity scores is shown in *Table 1*. The models are sorted from the most accurate to the least accurate. Higher cosine similarity values indicate a closer match between the generated code and the reference solution.

TABLE I. AVERAGE ACCURACY OF MODELS (BY SIMILARITY SCORE)

Model	*Cosine Similarity*
OpenAI GPT-4.5 Preview	0.89329872
OpenAI GPT-4o Mini	0.89225653
Claude 3 Haiku	0.87666922
OpenAI GPT-4o	0.87315825
Google Gemini 1.5 Flash	0.87104363
Claude 3 Sonnet	0.86526906
OpenAI GPT-o3 Mini	0.86520797
OpenAI GPT-o1	0.86216544
OpenAI GPT-4 Turbo	0.86158208
Google Gemini 1.5 Pro	0.85924462
OpenAI GPT-3.5 Turbo	0.85480591
Google Gemini 2.0 Flash Lite	0.84551985
Meta LLaMA 3.0 8B Instruct	0.83705625
Google Gemini 2.0 Flash	0.83415673
Claude 3 Opus	0.82712223
Claude 3.5 Sonnet	0.82435175
Claude 3.5 Haiku	0.82367317
Claude 3.7 Sonnet	0.82352193
Meta LLaMA 3.1 8B Instruct	0.7954627

The results indicate that *OpenAI GPT-4.5 Preview* performed the best in terms of similarity to the reference solutions, followed closely by *OpenAI GPT-4o Mini*. Conversely, *Meta LLaMA 3.1 8B Instruct* had the lowest average similarity score, indicating a higher deviation from the expected output.

B. Performance Across Simple and Complex Tasks

To analyze the impact of task complexity on model performance, we grouped the tasks into simple (Tasks 1-5) and complex (Tasks 6-10). The comparison of model accuracy across these two categories is presented in *Table 2*.

TABLE II. COMPARISON OF MODEL ACCURACY IN SIMPLE AND COMPLEX TASKS

Model	*Simple Tasks*	*Complex Tasks*
OpenAI GPT-4.5 Preview	0.86889777	0.91769967
OpenAI GPT-4o	0.86621171	0.8801048
Claude 3 Haiku	0.86393025	0.8894082
OpenAI GPT-4o Mini	0.86287462	0.92163844
Google Gemini 1.5 Pro	0.85128746	0.86720177
OpenAI GPT-o1	0.85042882	0.87390207
OpenAI GPT-o3 Mini	0.84759776	0.88281818
Claude 3 Opus	0.84163033	0.81261414
Claude 3 Sonnet	0.83993079	0.89060733
Claude 3.5 Haiku	0.83923439	0.80811194
Google Gemini 1.5 Flash	0.83543811	0.90664915
Claude 3.5 Sonnet	0.83350056	0.81520294
OpenAI GPT-4 Turbo	0.81862258	0.90454158
Claude 3.7 Sonnet	0.81714939	0.82989447
Meta LLaMA 3.0 8B Instruct	0.81175579	0.86235671
Meta LLaMA 3.1 8B Instruct	0.80849061	0.7824348
OpenAI GPT-3.5 Turbo	0.80721797	0.90239385
Google Gemini 2.0 Flash Lite	0.80056558	0.89047412
Google Gemini 2.0 Flash	0.79689259	0.87142087

The results reveal that models generally performed better on complex tasks compared to simple tasks. This trend is particularly evident for *OpenAI GPT-3.5 Turbo*, which experienced a significant decline in performance on simple tasks. This trend is unexpected, as complex tasks are generally assumed to be more challenging for AI models. One possible explanation is that complex tasks often involve longer and more structured text outputs, which may lead to higher cosine similarity scores. This suggests that models might generate responses that are more lexically and semantically aligned with reference solutions in longer outputs. This limitation should be considered when interpreting the results, as it may not necessarily indicate a higher reasoning capability but rather a tendency for models to produce more verbose and structured responses.

C. Visualization of Results

To illustrate the performance trends, the following visualizations are provided (Figure 1 – 4).

The bar chart (Figure 1) presents the overall accuracy of each model. The highest-performing models exhibit scores above 0.85, while the lower-performing models fall below 0.80.

The heatmap (Figure 2) highlights variations in accuracy across different tasks for each model. Warmer colors represent higher cosine similarity, while cooler colors indicate lower similarity. Certain tasks (e.g., Task 2) appear to be more challenging across all models.

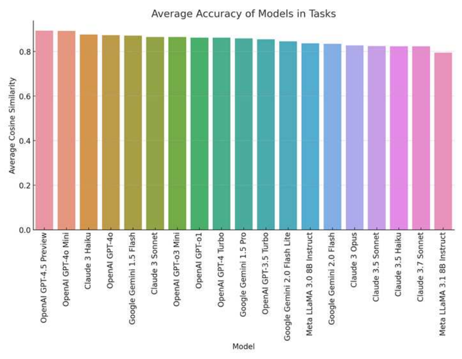

Figure 1. *Average Accuracy of Models in Tasks*

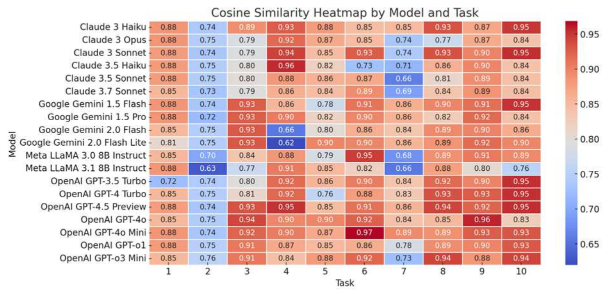

Figure 2. *Cosine Similarity Heatmap by Model and Task*

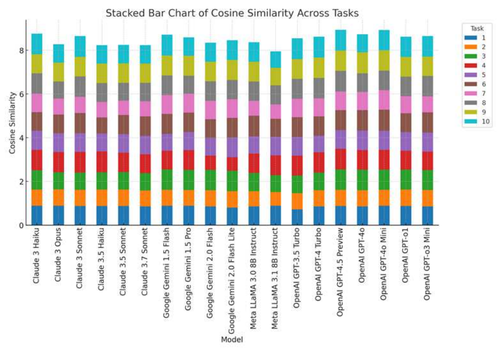

Figure 3. *Stacked Bar Chart of Cosine Similarity Across Tasks*

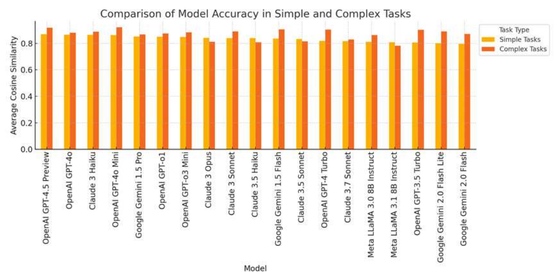

Figure 4. *Comparison of Model Accuracy in Simple and Complex Tasks*

The stacked bar chart (Figure 3) provides a comparative view of model performance across all tasks, showcasing how well models performed in each specific task.

The grouped bar chart (Figure 4) highlights the differences in model accuracy between simple and complex tasks. While some models maintain consistent accuracy across both categories, others struggle with more complex task structures.

D. *Summary of Findings*

From the experimental results, we can conclude the following:

The highest-performing models achieved cosine similarity scores exceeding 0.85, while lower-performing models struggled to reach 0.80.

Task complexity significantly impacts performance, with models generally performing better on complex tasks than on simple tasks.

Some models exhibit consistent performance across all task types, whereas others show substantial variation depending on task complexity.

Certain tasks are inherently more challenging, as indicated by lower similarity scores across all models.

Visualizations confirm the trends observed in the numerical data, providing a clearer understanding of model performance patterns.

These findings lay the foundation for further discussion on the implications of generative AI models for Python code generation.

VI. Discussion

The results presented in the previous section highlight key insights regarding the performance of generative AI models in Python code generation. This section discusses the observed trends, potential reasons behind variations in accuracy, and implications for practical use.

A. *Model Performance and Accuracy Trends*

The ranking (Table 1) shows consistent high performance from some models, with OpenAI GPT-4.5 Preview achieving the highest similarity (0.893). Conversely, Meta LLaMA 3.1 8B Instruct performed the worst (0.795). The stacked bar chart (Figure 3) confirms these variations, highlighting fluctuations in certain models.

B. *The Effect of Task Complexity*

Unexpectedly, complex tasks (Tasks 6-10) resulted in higher similarity scores than simpler ones (Tasks 1-5) (Table 2, Figure 4). This trend may stem from structured outputs in complex tasks, which align better with model predictions. The heatmap (Figure 2) shows that Task 2 posed challenges across models.

C. *Model-Specific Observations*

OpenAI GPT models demonstrated the strongest generalization across all tasks, making them the most versatile models.

Anthropic Claude models performed consistently well and handled various tasks effectively, though GPT models maintained an overall advantage in accuracy.

Google Gemini models showed balanced performance but had lower accuracy in all tasks.

Meta LLaMA models ranked lowest overall, with the newest 3.1 8B version performing the worst.

D. Key Findings and Implications

The findings suggest several important implications:

1. High-performing models generally excel across tasks, but differences in performance highlight the importance of careful model selection.
2. Contrary to expectations, complex tasks showed higher similarity, likely due to structured outputs rather than deeper reasoning.
3. Some newer models (e.g., Meta LLaMA 3.1 8B) underperformed, emphasizing the need for benchmarking before adoption.
4. Model strengths vary, suggesting that an ensemble approach may enhance reliability.
5. Cosine similarity is a useful baseline but should be complemented with execution correctness and coding best practices.

E. Limitations and Future Work

This study evaluates textual similarity but does not verify execution correctness. Prompt formulation impacts generation quality, requiring further research. Expanding the study to multiple programming languages and incorporating execution-based metrics would provide deeper insights.

VII. Conclusion

This study evaluated generative AI models for Python code generation using cosine similarity. Models like OpenAI GPT-4.5 Preview and GPT-4o Mini showed consistently high accuracy, while others varied with task complexity. Key findings include:

- Higher task complexity was linked to greater model accuracy.
- GPT models generalized best; Claude, Gemini, and LLaMA showed task-specific strengths.
- No single model excelled universally—hybrid or task-based selection may boost results.
- Cosine similarity is a solid baseline but should be complemented by functional and efficiency metrics.

Acknowledgment

The APC was supported by the SPEV 2025 Project run at the Faculty of Informatics and Management at the University of Hradec Kralove. Grammatical and stylistic revisions of this manuscript were assisted by the ChatGPT language model [17].

References

[1] Aarti, "Generative AI in Software Development: An Overview and Evaluation of Modern Coding Tools," IJFMR - International Journal For Multidisciplinary Research, vol. 6, no. 3, Jun. 2024. doi: 10.36948/ijfmr.2024.v06i03.23271.

[2] A. M. Dakhel, A. Nikanjam, F. Khomh, M. C. Desmarais, and H. Washizaki, "Generative AI for Software Development: A Family of Studies on Code Generation," in Generative AI for Effective Software Development, A. Nguyen-Duc, P. Abrahamsson, and F. Khomh, Eds. Cham: Springer, 2024, pp. 151–172. doi: 10.1007/978-3-031-55642-5_7.

[3] S. L. France, "Navigating Software Development in the ChatGPT and GitHub Copilot Era," Business Horizons, vol. 67, no. 5, pp. 649–661, Sep. 2024. doi: 10.1016/j.bushor.2024.05.009.

[4] A. Sajja, D. Thakur, and A. Mehra, "Integrating Generative AI into the Software Development Lifecycle: Impacts on Code Quality and Maintenance," International Journal of Science and Research Archive, vol. 13, no. 1, pp. 1952–1960, 2024. doi: 10.30574/ijsra.2024.13.1.1837.

[5] J. Sauvola, S. Tarkoma, M. Klemettinen, J. Riekki, and D. Doermann, "Future of Software Development with Generative AI," Automated Software Engineering, vol. 31, no. 1, p. 26, May 2024. doi: 10.1007/s10515-024-00426-z.

[6] B. Idrisov and T. Schlippe, "Program Code Generation with Generative AIs," Algorithms, vol. 17, no. 2, p. 62, Feb. 2024. doi: 10.3390/a17020062.

[7] D. Tosi, "Studying the Quality of Source Code Generated by Different AI Generative Engines: An Empirical Evaluation," Future Internet, vol. 16, no. 6, p. 188, Jun. 2024. doi: 10.3390/fi16060188.

[8] A. Sobo, M. Awes, B. Almas, and N. Polatidis, "Evaluating LLMs for Code Generation in HRI: A Comparative Study of ChatGPT, Gemini, and Claude," Applied Artificial Intelligence, vol. 39, no. 1, p. 2439610, Dec. 2025. doi: 10.1080/08839514.2024.2439610.

[9] R. Wang, J. Guo, C. Gao, G. Fan, C. Y. Chong, and X. Xia, "Can LLMs Replace Human Evaluators? An Empirical Study of LLM-as-a-Judge in Software Engineering," arXiv preprint, arXiv:2502.06193, Feb. 2025. doi: 10.48550/arXiv.2502.06193.

[10] A. Yadav, H. Beniwal, and M. Singh, "PythonSaga: Redefining the Benchmark to Evaluate Code Generating LLMs," in Findings of the Association for Computational Linguistics: EMNLP 2024, Miami, FL, Nov. 2024, pp. 17113–17126. doi: 10.18653/v1/2024.findings-emnlp.996.

[11] B. Baudry et al., "Generative AI to Generate Test Data Generators," IEEE Software, vol. 41, no. 6, pp. 55–64, Nov. 2024. doi: 10.1109/MS.2024.3418570.

[12] B. Wang, C.-C. J. Kuo, and H. Li, "Just Rank: Rethinking Evaluation with Word and Sentence Similarities," in Proc. 60th Annual Meeting of the Association for Computational Linguistics (ACL), Dublin, Ireland, May 2022, pp. 6060–6077. doi: 10.18653/v1/2022.acl-long.419.

[13] T. Mahmud, B. Duan, C. Pasareanu, and G. Yang, "Enhancing LLM Code Generation with Ensembles: A Similarity-Based Selection Approach," arXiv preprint, arXiv:2503.15838, Mar. 2025. doi: 10.48550/arXiv.2503.15838.

[14] J. C. Rollman et al., "Practical Design and Benchmarking of Generative AI Applications for Surgical Billing and Coding," arXiv preprint, arXiv:2501.05479, Jan. 2025. Available: https://arxiv.org/abs/2501.05479.

[15] G. Rani, J. Singh, and A. Khanna, "Comparative Analysis of Generative AI Models," in 2023 International Conference on Advances in Computation, Communication and Information Technology (ICAICCIT), Nov. 2023, pp. 760–765. doi: 10.1109/ICAICCIT60255.2023.10465941.

[16] S. Ramalakshmi and G. Asha, "Exploring Generative AI: Models, Applications, and Challenges in Data Synthesis," Asian Journal of Research in Computer Science, vol. 17, no. 12, pp. 123–136, Dec. 2024. doi: 10.9734/ajrcos/2024/v17i12533.

[17] OpenAI, "Chatgpt (gpt-4o)." https://chat.openai.com, 2025. Accessed 15 May 2025