Received 28 February 2025, accepted 8 April 2025, date of publication 11 April 2025, date of current version 18 April 2025.

Digital Object Identifier 10.1109/ACCESS.2025.3560244

RESEARCH ARTICLE

Evaluation of Generative AI Models in Python Code Generation: A Comparative Study

DOMINIK PALLA AND ANTONIN SLABY
Faculty of Informatics and Management, University of Hradec Kralove, 500 03 Hradec Kralove, Czech Republic

Corresponding author: Dominik Palla (dominik.palla@uhk.cz)

This work was supported by Article Processing Charge (APC) through the Specific Research Support Project 2025 (SPEV 2025) Run with the Faculty of Informatics and Management, University of Hradec Kralove.

ABSTRACT This study evaluates leading generative AI models for Python code generation. Evaluation criteria include syntax accuracy, response time, completeness, reliability, and cost. The models tested comprise OpenAI's GPT series (GPT-4 Turbo, GPT-4o, GPT-4o Mini, GPT-3.5 Turbo), Google's Gemini (1.0 Pro, 1.5 Flash, 1.5 Pro), Meta's LLaMA (3.0 8B, 3.1 8B), and Anthropic's Claude models (3.5 Sonnet, 3 Opus, 3 Sonnet, 3 Haiku). Ten coding tasks of varying complexity were tested across three iterations per model to measure performance and consistency. Claude models, especially Claude 3.5 Sonnet, achieved the highest accuracy and reliability. They outperformed all other models in both simple and complex tasks. Gemini models showed limitations in handling complex code. Cost-effective options like Claude 3 Haiku and Gemini 1.5 Flash were budget-friendly and maintained good accuracy on simpler problems. Unlike earlier single-metric studies, this work introduces a multi-dimensional evaluation framework that considers accuracy, reliability, cost, and exception handling. Future work will explore other programming languages and include metrics such as code optimization and security robustness.

INDEX TERMS Automatization, generative AI, LLM, python, software development.

I. INTRODUCTION

Generative AI models, particularly large language models (LLMs), have seen substantial advancements in recent years. These models—such as OpenAI's GPT series, Google's Gemini, and Meta's LLaMA—are capable of generating text, including programming code [1], [2]. Python's simplicity and readability have positioned it as a primary language for AI-driven code generation in areas like web development, data analysis, and machine learning [3].

The integration of generative AI in software development offers several notable benefits:

- **Increased Productivity:** Streamlining routine coding tasks.
- **Error Reduction:** Reducing human errors by promoting consistent coding standards.
- **Enhanced Accessibility:** Enabling non-experts to generate functional software.
- **Development Speed:** Accelerating the software lifecycle through quick code generation for specific tasks.

Generative AI models, like GPT-4, have shown significant value in automating coding tasks and improving the efficiency of Model-Driven Development (MDD) within Agile environments, as demonstrated by Sadik et al. [4]. However, while these foundational models offer great potential, they also bring unique risks, particularly in sensitive or critical applications [5].

Through experimentation, studies highlight how generative models show differences in algorithmic efficiency and performance under time-sensitive and precision-critical tasks [6].

Despite their advantages, generative AI models still face challenges in ensuring accuracy and reliability of the generated code [7]. This study extends prior research by incorporating a comprehensive evaluation framework that assesses not only accuracy but also reliability across multiple iterations, cost-effectiveness, and exception handling. Unlike earlier works that rely on single-metric assessments, our

The associate editor coordinating the review of this manuscript and approving it for publication was Zhiwu Li.

approach offers a holistic comparison, making it more applicable for real-world software development scenarios. This study aims to evaluate and compare the performance of leading generative AI models in Python code generation, addressing these ongoing challenges.

While AI-generated code can streamline development, studies highlight quality concerns in AI-modified code, as opposed to code generated from scratch, indicating the need for thorough quality checks prior to integration [8].

A. OBJECTIVES OF THE STUDY

The study's primary objective is to compare the generative AI models' abilities to generate Python code. This includes:

- Defining a **qualitative evaluation** scale for generated code to enable a more precise assessment of outputs.
- Designing **programming tasks** to evaluate diverse aspects of code generation, with each task executed across three iterations for each model to ensure consistency and reliability in results.
- **Analyzing the performance** of various AI models based on multiple criteria, including response time, accuracy, syntax correctness, code completeness, reliability across iterative tests, exception handling, time efficiency of the generated code, and cost-effectiveness.

II. THEORETICAL FRAMEWORK

A. UNDERSTANDING GENERATIVE AI MODELS

Generative AI models, particularly those based on large language models (LLMs), have revolutionized fields such as text generation and code synthesis. These models are built upon architectures like the Transformer model introduced by Vaswani et al. [1]. The Transformer architecture allows the model to capture long-range dependencies in text, making it highly effective for tasks such as code generation.

Key parameters influencing model performance include:

- **Temperature:** This parameter controls the randomness of the model's output by adjusting the distribution over the possible next tokens. Lower temperatures (e.g., 0.2–0.5) generate more deterministic and conservative responses, useful for factual or straightforward outputs. Higher temperatures (e.g., 0.8–1.2) introduce greater variability, enhancing creativity and diversity in the generated text [9].
- **Top-K Sampling:** Limits the choice of possible next words or tokens to the top K highest-probability candidates. By setting a fixed number, K, Top-K Sampling encourages the model to focus on the most likely options, minimizing less probable choices and generating coherent but occasionally constrained responses [10].
- **Top-P (Nucleus) Sampling:** Instead of a fixed count of candidates, Top-P Sampling (also known as Nucleus Sampling) dynamically selects the smallest set of tokens whose cumulative probability surpasses a specified threshold P. This approach offers a balance between focus and diversity, allowing the model to include some variability while avoiding highly unlikely choices [11].

In recent years, the AI-powered coding tools, including GitHub Copilot, OpenAI Codex (today outdated), and Amazon CodeGuru, have reshaped the development process by providing real-time code suggestions and intelligent completions [12].

B. APPLICATIONS AND IMPLICATIONS

The rapid advancements in generative artificial intelligence have unlocked transformative opportunities across diverse domains, including content creation, language translation, and software engineering. In particular, generative AI has shown considerable promise in code generation, reshaping development practices and enabling automated solutions that enhance productivity and efficiency. These developments underscore the broad applicability and impact of generative AI technologies, suggesting a future where intelligent systems play a crucial role in streamlining complex tasks and accelerating innovation [13], [14].

In the context of software development, these models can automate repetitive coding tasks, provide coding assistance, and even generate entire functions or modules based on high-level descriptions. This can lead to increased productivity, reduced errors, and greater accessibility for non-experts in coding [7], [15].

However, challenges remain in ensuring the accuracy, reliability, and maintainability of the generated code. Studies have highlighted the importance of rigorous evaluation and validation to assess the performance of AI-generated code and address potential shortcomings such as logical errors, security vulnerabilities, and inefficiencies [16], [17].

Generative AI models like ChatGPT are also being employed in educational contexts, providing formative feedback in programming courses, though accuracy and contextual understanding vary [18], [19].

They has also shown promising results in automating the generation of unit tests, a critical aspect of software validation. Recent studies highlight that tools like ChatGPT can effectively generate unit test cases for diverse Python code structures, including procedural scripts, function-based, and class-based code, offering significant time savings in test script creation [20].

In addition, AI models are being used for cross-language code translation, where results indicate that AI-assisted outputs, despite imperfections, lead to fewer errors when used with human oversight [21].

C. PYTHON CODE GENERATION

Python's readability and versatility make it an ideal language for code generation. Generative AI models are trained to understand syntax, semantics, and common Python programming patterns, enabling them to generate functional Python code with minimal human intervention [7].

III. METHODOLOGY

A. SELECTION OF AI MODELS

The study compares the performance of several prominent generative AI models in generating Python code. The models selected for this study are:

- **OpenAI Models:** GPT-4 Turbo, GPT-4o, GPT-4o Mini, GPT-3.5 Turbo.
- **Google Gemini Models:** Gemini 1.0 Pro, Gemini 1.5 Flash, Gemini 1.5 Pro.
- **Meta LLaMA Models:** LLaMA 3.0 8B, LLaMA 3.1 8B.
- **Anthropic Claude Models:** Claude 3.5 Sonnet, Claude 3 Opus, Claude 3 Sonnet, Claude 3 Haiku.

These models were selected based on their popularity, accessibility through APIs, and strong performance reputation in code generation tasks [7]. Additionally, insights from previous studies, such as those comparing Llama-2 and GPT-3 in complex computational tasks, help contextualize the capabilities of generative models in handling specialized coding challenges [22].

We need to emphasize the differences in generative models and provide practical insights into selecting the appropriate model for our task [23].

1) EXCLUSION OF CODE-SPECIFIC MODELS

In addition to the main generative AI models, there are also code-focused tools like GitHub Copilot and Amazon CodeGuru. These tools were excluded from our study for the following reasons:

- **GitHub Copilot:** GitHub Copilot relies on OpenAI's models, specifically Codex and now GPT-3.5 Turbo, which are already represented in our selection. Including GitHub Copilot would introduce redundancy, as it essentially serves as an interface for OpenAI's underlying models rather than a distinct model with unique capabilities [24].
- **Amazon CodeGuru:** Amazon CodeGuru focuses more on code quality analysis and review, aiming to identify performance improvements and potential errors rather than generating new code from scratch. As such, its functionality differs from the generative AI models selected for this study, which are designed to produce original code based on a prompt [25].

By focusing on models specifically designed for generative tasks, this study provides a clearer understanding of the capabilities and performance trade-offs among leading AI-driven code generation solutions.

B. MODEL OVERVIEW AND DIFFERENCES

Each generative AI model used in this study has distinct strengths and weaknesses. This section provides a comparative overview of the models tested in terms of their intended use cases, cost, and performance characteristics.

1) OPENAI MODELS

OpenAI offers a range of models, each designed for different levels of performance and cost-effectiveness:

- **GPT-4 Turbo:** A robust variant of the GPT-4 model, optimized for high accuracy, detailed responses, and complex code generation tasks. It is ideal for scenarios requiring high-quality output but comes with higher costs [26].
- **GPT-4o:** An optimized version of GPT-4, designed to be more cost-effective and slightly faster while still maintaining strong code generation performance. It is suitable for cases where budget and speed are priorities [27].
- **GPT-4o Mini:** A streamlined variant that provides faster responses at a significantly reduced cost, though with trade-offs in complexity and detail, making it suitable for less demanding tasks [28].
- **GPT-3.5 Turbo:** A well-established predecessor to the GPT-4 models, it remains effective for simpler coding tasks and offers a lower-cost option for budget-conscious applications [26].

OpenAI's **Codex model**, introduced in 2021, was specifically designed for code generation tasks. However, as of March 2023, OpenAI discontinued support for the Codex API, advising users to transition to the more advanced GPT-3.5 Turbo model for coding applications [29]. This shift reflects the rapid advancements in AI models, with newer iterations like GPT-3.5 Turbo and GPT-4 offering enhanced capabilities and performance over the now-outdated Codex.

2) GOOGLE GEMINI MODELS

Google Bard, initially introduced as a conversational AI service, has undergone significant evolution. In September 2023, Google announced that Bard would be rebranded as **Gemini**, reflecting its integration with Google's latest AI advancements. This transition signifies a shift towards more advanced capabilities and a broader range of applications [30].

The rebranding to Gemini aligns with Google's strategy to enhance its AI offerings, incorporating more sophisticated models and features. This move aims to provide users with more powerful and versatile AI tools, expanding beyond the original scope of Bard.

As a result, the original Bard service has been deprecated in favor of the more advanced Gemini platform. Users and developers are encouraged to transition to Gemini to leverage its enhanced capabilities and stay aligned with Google's latest AI developments [31].

Google's **Gemini** models provide an alternative suite of generative models, each with a specific focus on either performance or cost:

- **Gemini 1.0 Pro:** A robust model designed for complex tasks with a high degree of accuracy, but at a relatively higher cost. It is optimized for generating code that requires precision [32].
- **Gemini 1.5 Flash:** A faster, more efficient model aimed at reducing both response time and costs. Ideal for simpler tasks that don't require the same depth of complexity [33].

- **Gemini 1.5 Pro:** A high-end model designed for both accuracy and speed. It's an evolution of the Gemini 1.0 Pro, offering improved performance but with higher costs [34].

3) META LLAMA MODELS

Meta's LLaMA models focus on providing high-quality text generation at lower costs, particularly when compared to other high-performance models:

- **LLaMA 3.0 8B:** This model balances performance and cost, delivering high-quality code generation suitable for a wide range of tasks. It features 8 billion parameters and is designed to be efficient for various applications [35].
- **LLaMA 3.1 8B:** An updated version of the LLaMA series, LLaMA 3.1 8B provides improvements in accuracy and performance over the 3.0 version while maintaining cost-effectiveness. It continues to offer 8 billion parameters and is optimized for multilingual dialogue use cases [36].

These models are part of Meta's commitment to advancing open-source AI, providing accessible and efficient tools for developers and researchers [37].

4) ANTHROPIC CLAUDE MODELS

Anthropic's Claude models provide different configurations designed to meet various needs in terms of cost, speed, and code generation complexity:

- **Claude 3.5 Sonnet:** A streamlined model designed to handle less complex tasks with a focus on faster response times and reduced costs, making it ideal for applications that prioritize speed and affordability over depth [38].
- **Claude 3 Opus:** A more versatile model that offers a balance between cost efficiency and performance. It maintains good accuracy for a range of coding tasks and is suitable for projects that require more precision than the Sonnet series, though at a slightly higher cost [39].
- **Claude 3 Sonnet:** Another model within the Sonnet series, optimized for quick responses at a low cost, designed also to efficiently handle simpler coding tasks [39].
- **Claude 3 Haiku:** The most lightweight model, offering the lowest cost and fastest response times in the series, making it ideal for the simplest tasks [40].

The Opus models provide higher accuracy and versatility, making them more suitable for complex tasks, while the Sonnet models prioritize speed and cost-effectiveness for simpler tasks [39].

C. EVALUATION CRITERIA

A qualitative scale was established to evaluate the generated code based on the following parameters:

- **Syntax Correctness:** Rating as YES (syntax completely correct) or NO (syntax incorrect). The syntax was verified using the PyCharm IDE's built-in code inspection tools [41].
- **Code Completeness:** Rating as YES (code is standalone fully functional) or NO (code is incomplete). Each code snippet tested by running (without any other enhancement).
- **Response Time:** Measured as the time from the API request to receiving the code. Each model tested on exact same server with the same internet connectivity.
- **Accuracy:** Rating from 5 (fully meets requirements) to 1 (does not meet requirements), or 0 (not working even that syntax is correct). Evaluated on the basis of a subjective comparison of the assignment and the result.
- **Reliability:** Rated on a scale from 5 (highly reliable) to 1 (unreliable). This metric assesses the model's consistency in generating similar solutions for the same task across three iterations. A rating of 5 indicates that all responses used the same approach. A rating of 3 is given if two responses used the same approach while one differed, and a rating of 1 is assigned when all three responses employed different approaches. Intermediate ratings of 2 and 4 are applied for minor subjective differences in approach.
- **Exception-Handling:** Rated as YES (exception handling present) or NO (no exception handling). Models were not explicitly prompted to include exception handling, so this criterion evaluates whether they independently incorporated any exception-handling mechanisms in the generated code.
- **API Usage Cost:** Total cost per request, calculated based on the input and output token usage in USD.
- **Time Efficiency:** The execution time of each generated script, allowing for comparison of different solution approaches. Those scripts which are not working do not have this value.

Evaluation criteria like those defined above are essential for assessing model performance in code generation tasks, as demonstrated in studies focused on code-specific language models [42], [43], [44], [45], [46], [47].

D. DESIGN OF PROGRAMMING TASKS

The study includes ten programming tasks of progressively increasing complexity, divided into simpler (1-5) and complex tasks (6-10). These tasks are designed to test different aspects of the models' code-generation capabilities. The selected tasks focus on Python-based general-purpose coding problems to ensure consistency in evaluation. More domain-specific tasks, such as SQL query generation or optimization-based coding, are left for future research.

1) SIMPLER TASKS

- Task 1: Hello World
- Task 2: Sum of Numbers in a List
- Task 3: Factorial of a Number
- Task 4: Palindrome Check

- Task 5: Fibonacci Sequence Calculation

2) COMPLEX TASKS

- Task 6: File Reading and Writing
- Task 7: Statistical Analysis of a Dataset
- Task 8: Web Scraping
- Task 9: Data Visualization
- Task 10: Simple Web Application

E. FORMULATION OF PROMPTS

For each programming task, unambiguous prompts were created.

- **Task 1:** Hello World:

```
"Write a simple program that prints the
    text 'Hello, World!' in Python."
```

- **Task 2:** Sum of Numbers in a List:

```
"Write a function that takes a list of
    numbers and returns their sum in
    Python."
```

- **Task 3:** Factorial of a Number:

```
"Write a function that calculates the
    factorial of a given number using
    recursion in Python."
```

- **Task 4:** Palindrome Check:

```
"Write a function that checks if a given
    string is a palindrome in Python."
```

- **Task 5:** Fibonacci Sequence Calculation:

```
"Write a function that returns a list of
    the first n numbers in the Fibonacci
    sequence in Python."
```

- **Task 6:** File Reading and Writing:

```
"Write a program that reads data from a
    text file and writes it to a new file,
     numbering each line in Python."
```

- **Task 7:** Statistical Analysis of a Dataset:

```
"Write a program that loads a dataset (CSV
     file) using the pandas library and
    performs basic statistical analysis (
    mean, median, standard deviation) on
    columns containing numerical data in
    Python."
```

- **Task 8:** Web Scraping:

```
"Write a script that downloads an HTML
    page from a given URL and extracts all
     links (tags <a>) using the
    BeautifulSoup library in Python."
```

- **Task 9:** Data Visualization:

```
"Write a program that loads a dataset (CSV
     file) using pandas and creates graphs
     (e.g., bar chart, histogram) using
    the Matplotlib or Seaborn library in
    Python."
```

- **Task 10:** Simple Web Application:

```
"Write a basic web application using the
    Flask framework that has a simple form
     for entering text and displays the
    entered text back on the page upon
    form submission in Python."
```

F. ADDITIONAL TRIMMING PROMPT

Each prompt instructs the model to generate Python code without comments or embedded text, ensuring that the API returns only executable code. To ensure consistency across all model comparisons, we standardized the API request parameters, setting Temperature to 0.7 and Top-P to 0.9. These values provide a balance between deterministic and creative outputs, ensuring comparability between different generative AI models.

```
"Give me only the executable code, no comments, no
    explanations, no other information, just the
    code."
```

G. API COMMUNICATION CODE EXAMPLES

Below are examples of API communication for each model:

1) OPENAI API (GPT-4 TURBO, GPT-4O MINI, GPT-3.5 TURBO)

```
# Token Usage Calculation
def calculate_tokens(text, model="gpt-4-turbo"):
    encoding = tiktoken.encoding_for_model(model)
    return len(encoding.encode(text))

def get_openai_code(prompt, model="gpt-4-turbo",
    input_cost_per_token=0.01,
    output_cost_per_token=0.03):
    start_time = time.time()

    response = openai.ChatCompletion.create(
        model=model,
        messages=[{'role': 'user', 'content':
            prompt}],
        max_tokens=150,
        temperature=0.7
    )

    end_time = time.time()

    generated_code = response['choices'][0]['
        message']['content']

    input_tokens = calculate_tokens(prompt, model)
    output_tokens = calculate_tokens(
        generated_code, model)

    total_cost = (input_tokens *
        input_cost_per_token) + (output_tokens *
        output_cost_per_token)

    return generated_code, end_time - start_time,
        total_cost
```

2) GOOGLE GEMINI API (GEMINI 1.0 PRO, 1.5 FLASH, 1.5 PRO)

```
def get_gemini_code(prompt, model="gemini-1.5-
    flash", input_cost_per_token=0.0015,
    output_cost_per_token=0.0025):
    start_time = time.time()
    try:
        gemini_model = genai.GenerativeModel(model
            )
        response = gemini_model.generate_content(
            prompt)
        generated_code = response.text

        # Token Usage Calculation
        input_tokens = calculate_tokens(prompt)
        output_tokens = calculate_tokens(
            generated_code)
        total_cost = (input_tokens *
            input_cost_per_token) + (output_tokens
             * output_cost_per_token)
    except Exception as e:
        return None, None, None
    end_time = time.time()

    return generated_code, end_time - start_time,
        total_cost
```

3) META LLAMA API (LLAMA 3.0 8B, LLAMA 3.1 8B)

```
# Token Usage Calculation
def calculate_tokens_transformers(text, model="
    meta-llama/Meta-Llama-3-8B-Instruct"):
    tokenizer = AutoTokenizer.from_pretrained(
        model)
    return len(tokenizer.encode(text))

def get_llama_code(prompt, model="meta-llama/Meta-
    Llama-3-8B-Instruct", input_cost_per_token
    =0.0028, output_cost_per_token=0.0028):
    api_url = f"https://api-inference.huggingface.
        co/models/{model}"
    headers = {'Authorization': f'Bearer {
        HUGGINGFACE_API_KEY}', 'Content-Type': '
        application/json'}
    data = {'inputs': prompt, 'parameters': {'
        temperature': 0.7, 'max_new_tokens': 150}}

    start_time = time.time()
    response = requests.post(api_url, headers=
        headers, json=data)
    end_time = time.time()

    response_data = response.json()
    generated_code = response_data[0].get('
        generated_text', "")
    total_tokens = calculate_tokens_transformers(
        prompt + generated_code, model)
    total_cost = total_tokens *
        input_cost_per_token

    return generated_code, end_time - start_time,
        total_cost
```

4) ANTHROPIC CLAUDE API (SONNET 3.5, SONNET 3, OPUS 3, HAIKU 3)

Unlike other models that use subword tokenization (where tokens are fragments of words), Anthropic's Claude models estimate tokens based on character count. Specifically, one token is approximately equal to four characters of text. This approximation helps in calculating token usage without complex tokenization algorithms [48].

```
# Token Usage Calculation
def calculate_claude_tokens(text):
    return len(text.split())

def get_claude_code(prompt, model="claude-3-5-
    sonnet-20241022", input_cost_per_token=0.002,
    output_cost_per_token=0.002):
    start_time = time.time()
    try:
        message = ant_client.messages.create(
            model=model, max_tokens=300,
                temperature=0.7,
            messages=[{"role": "user", "content":
                prompt}]
        )
        generated_code = message.content
        input_tokens = calculate_claude_tokens(
            prompt)
        output_tokens = calculate_claude_tokens(
            generated_code)
        total_cost = (input_tokens *
            input_cost_per_token) + (output_tokens
             * output_cost_per_token)
    except Exception as e:
        return None, None, None
    end_time = time.time()

    return generated_code, end_time - start_time,
        total_cost
```

H. TESTING SETUP AND TOOLS

Each prompt was evaluated across all models in three iterations to assess consistency and establish a baseline for a more detailed analysis. The repeated testing provides an average performance measure, supporting robust comparisons among models.

1) INFRASTRUCTURE SETUP

- **VPS Server:** A Linux-based VPS server (Debian 11 OS, 3 GB RAM, Intel(R) Xeon(R) E-2336 CPU @ 2.90GHz) was set up to ensure a stable environment for testing across multiple iterations. The server will send API requests and log the results for further analysis.
- **Software Environment:** Python 3 was installed on the server along with necessary libraries.

2) TOOLS AND LIBRARIES

- **Python:** Used to handle API calls, process responses, and automate testing.
- **IntelliJ PyCharm IDE:** The IntelliJ PyCharm Integrated Development Environment (IDE) was used to interactively run experiments, write and test code, and document the results efficiently. PyCharm's robust debugging and project management features streamlined the testing process [49].
- **Requests Library:** The Requests library was utilized to manage HTTP requests to each model's API endpoint, enabling reliable and consistent communication with the AI models for each test prompt [50].

- **Pandas Library:** Pandas was essential for organizing, structuring, and storing the test results in a tabular format, facilitating subsequent data analysis and comparisons. This library's powerful data manipulation capabilities made it a core component of the data processing pipeline [51].
- **Matplotlib and Plotly Libraries:** Both Matplotlib and Plotly were used to visualize the collected data, providing clear and insightful charts to facilitate a better comparison of model performance across various metrics. Matplotlib allowed for static plots, while Plotly added interactivity to the visualizations [52].
- **Tiktoken Library:** Tiktoken was employed for tokenization tasks, particularly to manage token counting and ensure accurate usage tracking for OpenAI models, helping to calculate API usage costs [53].
- **Transformers Library:** The Hugging Face Transformers library was used to facilitate interactions with models compatible with the Transformers framework. This library provided the necessary tools for handling model outputs, preprocessing prompts, and interpreting results effectively [54].

3) PRICING OF MODELS

Each model has its own pricing structure, typically with different rates for input and output tokens. To efficiently manage and calculate the costs, it is necessary to maintain a comprehensive list of relevant attributes for each model, including the function used to call the model and the cost per token for both input and output.

Upon completion of code generation, the number of tokens utilized is counted, and the corresponding cost is recorded.

Below is shown how this information is structured for each model:

```
# List of models with associated costs per token (
    in USD)
models = [
    {"name": "OpenAI GPT-4 Turbo", "function":
        get_openai_code, "model_param": "gpt-4-
        turbo",
    "input_cost_per_token": 0.01/1000, "
        output_cost_per_token": 0.03/1000},
    {"name": "OpenAI GPT-4o", "function":
        get_openai_code, "model_param": "gpt-4o",
    "input_cost_per_token": 0.005/1000, "
        output_cost_per_token": 0.015/1000},
    {"name": "OpenAI GPT-4o Mini", "function":
        get_openai_code, "model_param": "gpt-4o-
        mini",
    "input_cost_per_token": 0.00015/1000, "
        output_cost_per_token": 0.0006/1000},
    {"name": "OpenAI GPT-3.5 Turbo", "function":
        get_openai_code, "model_param": "gpt-3.5-
        turbo",
    "input_cost_per_token": 0.003/1000, "
        output_cost_per_token": 0.006/1000},
    {"name": "Google Gemini 1.0 Pro", "function":
        get_gemini_code, "model_param": "gemini
        -1.0-pro",
     "input_cost_per_token": 0.0005/1000, "
         output_cost_per_token": 0.0015/1000},
    {"name": "Google Gemini 1.5 Flash", "function"
        : get_gemini_code, "model_param": "gemini
        -1.5-flash",
     "input_cost_per_token": 0.000075/1000, "
         output_cost_per_token": 0.0003/1000},
     {"name": "Google Gemini 1.5 Pro", "function":
         get_gemini_code, "model_param": "gemini
         -1.5-pro",
     "input_cost_per_token": 0.0035/1000, "
         output_cost_per_token": 0.0105/1000},
    {"name": "Meta LLaMA 3.0 8B", "function":
        get_llama_code, "model_param": "meta-llama
        /Meta-Llama-3-8B-Instruct",
    "input_cost_per_token": 0.0028/1000, "
        output_cost_per_token": 0.0028/1000},
    {"name": "Meta LLaMA 3.1 8B", "function":
        get_llama_code, "model_param": "meta-llama
        /Meta-Llama-3.1-8B-Instruct",
    "input_cost_per_token": 0.0028/1000, "
        output_cost_per_token": 0.0028/1000},
    {"name": "Claude 3.5 Sonnet", "function":
        get_claude_code, "model_param": "claude
        -3-5-sonnet-latest",
     "input_cost_per_token": 0.003 / 1000, "
         output_cost_per_token": 0.015 / 1000},
    {"name": "Claude 3 Opus", "function":
        get_claude_code, "model_param": "claude-3-
        opus-latest",
     "input_cost_per_token": 0.015 / 1000, "
         output_cost_per_token": 0.075 / 1000},
    {"name": "Claude 3 Sonnet", "function":
        get_claude_code, "model_param": "claude-3-
        sonnet-20240229",
     "input_cost_per_token": 0.003 / 1000, "
         output_cost_per_token": 0.015 / 1000},
    {"name": "Claude 3 Haiku", "function":
        get_claude_code, "model_param": "claude-3-
        haiku-20240307",
     "input_cost_per_token": 0.00025 / 1000, "
         output_cost_per_token": 0.00125 / 1000},
]
```

I. ITERATIVE TESTING PROCEDURE

1) STEP-BY-STEP TESTING PROCESS

1) **Preparation of Testing Scripts:** Dedicated Python scripts were created for each model described in chapter III. to manage API calls, as outlined in section III-G.
2) **Automated Execution:** A main script was developed to automate the entire testing workflow for each task and model. The automated process included the following steps:
    - **Send Prompt**: The current task prompt was sent to the model's API.
    - **Measure Response**: Response time from the API call was recorded, and token usage was tracked to calculate the associated cost.
    - **Save Generated Code**: The generated code, along with response time and other relevant data, was saved for analysis.
    - **Data Collection**: Key data points, including response time, token usage, and code output, were collected for each test iteration to ensure a comprehensive dataset.
    - **Testing Iterations**: Each task was tested three times per model to assess consistency and

minimize variability. In total, each model was tested on 10 tasks with 3 iterations per task, resulting in 30 tests per model.

- **Exception Handling**: If an API request failed or returned incomplete results, the error was logged, and the test was automatically retried to maintain data consistency.
- **Log Results**: All results from each test were stored in a CSV file, providing a structured dataset for subsequent analysis.
- **Qualitative Evaluation**: The generated code was evaluated according to the study's criteria to assess the quality of the response.
- **Analysis and Visualization**: Data was exported, and various metrics were calculated. Visualizations, including tables and charts, were created for model comparison.

2) SIMPLIFIED VERSION OF THE MAIN SCRIPT

The complete main script spans approximately 300 lines of code. Below is a streamlined version to illustrate its key functions and flow:

```
import time
import pandas as pd
import logging
from openai_test import get_openai_code
from gemini_test import get_gemini_code
from llama_test import get_llama_code
from claude_test import get_claude_code

# List of tasks and model configurations
tasks = ["Task 1 prompt", "Task 2 prompt", ...]
# Define all tasks
models = [
    {"name": "OpenAI GPT-4 Turbo", "function":
        get_openai_code},
    {"name": "Google Gemini 1.5 Flash", "function"
        : get_gemini_code},
    {"name": "Meta LLaMA 3.0 8B", "function":
        get_llama_code},
    {"name": "Claude 3.5 Sonnet", "function":
        get_claude_code}
]
# Define all models
results = []

for task in tasks:
    for model in models:
        for iteration in range(3):
            start_time = time.time()
            try:
                result_text, response_time,
                    total_cost = model["function"
                    ](task)
                duration = time.time() -
                    start_time
                results.append({
                    "Model": model["name"],
                    "Task": task,
                    "Iteration": iteration + 1,
                    "Response Time (s)":
                        response_time,
                    "Generated Code": result_text,
                    "API Cost (USD)": total_cost
                })
            except Exception as e:
                results.append({
                    "Model": model["name"],
                    "Task": task,
                    "Iteration": iteration + 1,
                    "Error": str(e)
                })
# Save results to CSV
pd.DataFrame(results).to_csv("api_test_results.csv
    ", index=False)
```

3) EXECUTION AND VALIDATION MODULE

The generated code is executed to test for runtime errors and validate functionality. The time taken to execute the code is also measured to provide insights into the efficiency of each generated script. The following Python function handles execution, error handling, and timing:

- **IDE syntax check:** PyCharm's syntax checker points out all the syntax-error parts using its code inspection mechanisms [41].
- **Running test:** We run the generated code and wait for some error printouts:

```
import time

def execute_code(code):
    start_time = time.time()
    try:
        exec(code)
        execution_time = time.time() -
            start_time
        return True, execution_time
    except Exception as e:
        execution_time = time.time() -
            start_time
        print(f"Execution error: {e}")
        return False, execution_time
```

In this function:

- The code is executed within a `try-except` block to catch any runtime exceptions.
- The `start_time` and `execution_time` variables are used to measure the execution duration.
- If the code runs without errors, the function returns `True` and the `execution_time`.
- If an exception occurs, it returns `False` along with the `execution_time`, and the error is printed for debugging purposes.

This automated approach applied to all generated code snippets ensured that each snippet's validation results, including execution success and time, were systematically saved and exported as a CSV file for further analysis.

IV. RESULTS

This section presents the performance results of the generative AI models on ten Python coding tasks. The complete dataset comprises approximately 350 rows.

A. ADDITIONAL EVALUATION CRITERIA

Before proceeding with the full analysis, it is necessary to incorporate further evaluation criteria that were defined earlier in the article. These criteria include:

- Syntax Correctness
- Code Completeness
- Accuracy
- Reliability
- Exception Handling
- Time Efficiency

These values will be filled in based on manual assessment and tools such as PyCharm for syntax checking, as well as functional testing of the generated code (as described in chapter III). Once these metrics are gathered, the complete analysis will be performed.

B. TOP MODELS IN API RESPONSE TIME

Table 1 presents the top 3 models in terms of the fastest API response times (average of 3 iterations). It displays the top 3 models for simple tasks, complex tasks, and all tasks combined.

The fastest are models Claude 3 Haiku and Gemini 1.5 Flash.

The slowest response time was recorded for the Gemini 1.5 Pro model (almost 4 seconds per request - 3.99). As table 1 presents, GPT-4-o-mini was excluded in favor of GPT-4-o for more complex tasks due to its slower performance.

C. TOP AND WORST MODELS IN API COST

Table 2 presents the models with the lowest API costs, while Table 3 highlights those with the highest costs. The most cost-effective model is Gemini 1.5 Flash, while the highest average cost was recorded for the Claude Opus 3 model.

D. SYNTAX CORRECTNESS AND CODE COMPLETENESS

All models exhibited correct syntax.

Some models, including GPT-3.5 Turbo, Gemini 1.0 Pro, Gemini 1.5 Flash, Meta LLaMA 3.0, and Meta LLaMA 3.1, consistently failed to produce complete code for Task 10 across all iterations (they did not include the view part of the code, that is why it could not be run).

E. EXCEPTION HANDLING

Only the Claude 3.5 Sonnet model successfully implemented exception handling - for Task 8 in all three iterations. No other exception handling was observed. The lack of exception-handling mechanisms in most AI-generated code samples raises concerns about robustness in real-world applications. Future improvements in AI-assisted coding should prioritize automated generation of robust error-handling mechanisms, particularly in security-sensitive environments.

F. ACCURACY

Tables 4 and 5 show the top 6 models ranked by accuracy.

The results highlight Claude 3.5 Sonnet and Claude 3 Sonnet as top performers in accuracy across all task types. For simple tasks, these models achieved perfect accuracy, while Claude 3 Opus also demonstrated strong performance with a score of 4.93. In complex tasks, all Claude models achieved a perfect 5.00 score, indicating their robust handling of higher difficulty levels. OpenAI's GPT models, including GPT-4o and GPT-4 Turbo, performed well, particularly in complex tasks, showcasing their capability in accuracy for more demanding prompts.

The lowest accuracy was observed in the Gemini 1.0 Pro model, with an overall score of 4.37 and a score of 4.13 for simple tasks.

G. RELIABILITY

Table 6 presents the top 3 models in terms of reliability for overall tasks, simple tasks, and complex tasks.

For simple tasks, Claude 3 Sonnet, Meta LLaMA 3.1 8B, and Meta LLaMA 3.0 8B achieved a flawless reliability score of 5.0. For complex tasks, Claude 3.5 Sonnet, Meta LLaMA 3.1 8B, and Claude 3 Sonnet demonstrated the highest reliability. Overall, Claude 3 Sonnet and Meta LLaMA 3.1 8B stood out as the most consistent performers across all tasks.

The lowest reliability was recorded for the Gemini 1.0 Pro model, with an overall score of 3.2 and a score of 3.0 for simple tasks. In comparison, even the smaller model, GPT-4o Mini, achieved an overall score of 3.80.

H. TIME EFFICIENCY

Table 7 highlights the top 3 models in terms of time efficiency. Although the performance of the tested models was relatively similar (with a few exceptions), this section focuses on those that achieved the shortest execution times.

I. VISUAL SUMMARY AND COMPARISON

To provide a comprehensive and comparative overview of the generative AI models' performance, Figures 1 and 2 present visual summaries of key metrics: Accuracy, Reliability, API Response Time, API Cost, and Time Effectiveness. These figures consolidate findings across all ten programming tasks, offering insights into each model's strengths and weaknesses in handling diverse code generation requirements.

Figure 1 employs a radar chart to give a clear overview of each model's capabilities. This visual structure helps readers quickly identify top-performing models for various use cases, such as high-accuracy models for precision-demanding tasks or models with lower API costs for budget-sensitive applications.

To further clarify the performance evaluation, Figure 2 provides an alternative radar chart with inverted scales for metrics where lower values represent better performance. Specifically, for **API Response Time**, **API Cost**, and **Time Effectiveness**, the scales are adjusted so that lower values (indicating faster response times, lower costs, and higher efficiency) appear as higher scores. This inversion aligns with the idea that higher scores signify better performance across all metrics, making comparisons more intuitive.

Together, these figures offer a side-by-side comparison of the models, highlighting how they perform under both the original and adjusted scales. This visual summary serves as a practical reference for developers and researchers aiming to select the optimal model based on task complexity, performance requirements, and cost considerations. The

TABLE 1. Top 3 models by response time for simple (Tasks 1-5), complex (Tasks 6-10) and all tasks.

Tasks	1st Best Model	2nd Best Model	3rd Best Model
Simple	Claude 3 Haiku (0.66s)	Google Gemini 1.5 Flash (0.67s)	OpenAI GPT-4o Mini (0.73s)
Complex	Google Gemini 1.5 Flash (1.02s)	Claude 3 Haiku (1.26s)	OpenAI GPT-4o (1.34s)
All	Google Gemini 1.5 Flash (0.85s)	Claude 3 Haiku (0.96s)	OpenAI GPT-4o (1.05s)

TABLE 2. Top 3 models by API cost (the cheapest ones).

1st Best Model	2nd Best Model	3rd Best Model
Google Gemini 1.5 Flash (0.000022$)	Claude 3 Haiku (0.000042$)	OpenAI GPT-4o Mini (0.000045$)

TABLE 3. Worst 3 models by API cost (the most expensive ones).

1st Best Model	2nd Best Model	3rd Best Model
Claude 3 Opus (0.0023$)	OpenAI GPT-4 Turbo (0.0022$)	OpenAI GPT-4o (0.0011$)

TABLE 4. Top 3 models by accuracy in responses for simple (Tasks 1-5), complex (Tasks 6-10), and all tasks.

Tasks	1st Best Model	2nd Best Model	3rd Best Model
Simple	Claude 3.5 Sonnet - Claude 3 Sonnet (5.00)		Claude 3 Opus (4.93)
Complex	Claude 3.5 Sonnet - Claude 3 Sonnet - Claude 3 Opus (5.00)		
All	Claude 3.5 Sonnet (5.00) - Claude 3 Sonnet (5.00)		Claude 3 Opus (4.97)

TABLE 5. 4th-6th place in models by accuracy in responses for simple (Tasks 1-5), complex (Tasks 6-10) and all tasks.

Tasks	4th Best Model	5th Best Model	6th Best Model
Simple	Google Gemini 1.5 Pro - Claude 3 Haiku (4.93)		OpenAI GPT-4 Turbo - GPT 3.5 Turbo (4.87)
Complex	OpenAI GPT-4o - GPT-4 Turbo - GPT-4o Mini (5.00)		
All	OpenAI GPT-4 Turbo - GPT-4o Mini (4.93)		OpenAI GPT-4o - Gemini 1.5 Pro (4.90)

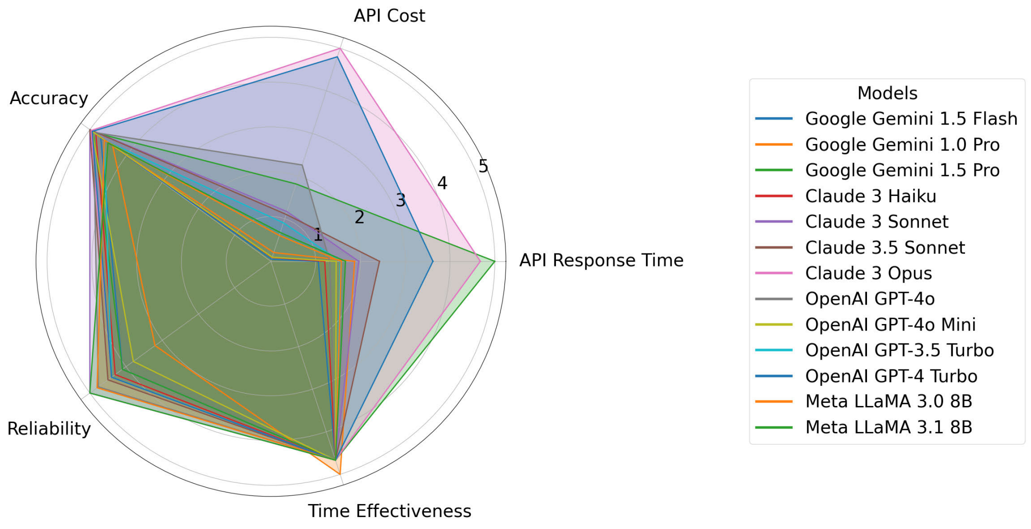

FIGURE 1. Comprehensive Comparison of Model Capabilities, matplotlib librar.

TABLE 6. Top 3 models by reliability in responses for simple (Tasks 1-5), complex (Tasks 6-10), and all tasks.

Tasks	1st Best Model	2nd Best Model	3rd Best Model
Simple	Claude 3 Sonnet - Meta LLaMA 3.1 8B - Meta LLaMA 3.0 8B (5.00)		
Complex	Claude 3.5 Sonnet - Meta LLaMA 3.1 8B - Claude 3 Sonnet (5.00)		
All	Claude 3 Sonnet - Meta LLaMA 3.1 8B (5.00)		Meta LLaMA 3.0 8B (4.80)

inclusion of both figures underscores the trade-offs between cost and performance, guiding readers in choosing the best model to suit specific project needs.

The plots were generated using Python's matplotlib library [52], employing radar charts to illustrate differences among models across multiple criteria in an accessible and visually informative manner.

V. DISCUSSION

This study reveals clear differences in how generative AI models perform in Python code generation. While all models handled syntax correctly, they varied significantly in how complete, consistent, efficient, or robust their outputs were. These differences have practical implications, especially in choosing the right model for a specific use case.

TABLE 7. Top 3 models by time efficiency of the generated python code for simple (Tasks 1-5), complex (Tasks 6-10) and all tasks.

Tasks	1st Best Model	2nd Best Model	3rd Best Model
Simple	Google Gemini 1.0 Pro (1.16e-05s)	Meta LLaMA 3.0 8B (1.19e-05s)	Google Gemini 1.5 Flash (1.19e-05s)
Complex	Meta LLaMA 3.1 8B (2.85s)	Claude 3 Sonnet (2.85+s)	Meta LLaMA 3.0 8B (2.85++)
All	Meta LLaMA 3.1 8B (1.42s)	Claude 3 Sonnet (1.42+s)	Meta LLaMA 3.0 8B (1.42++s)

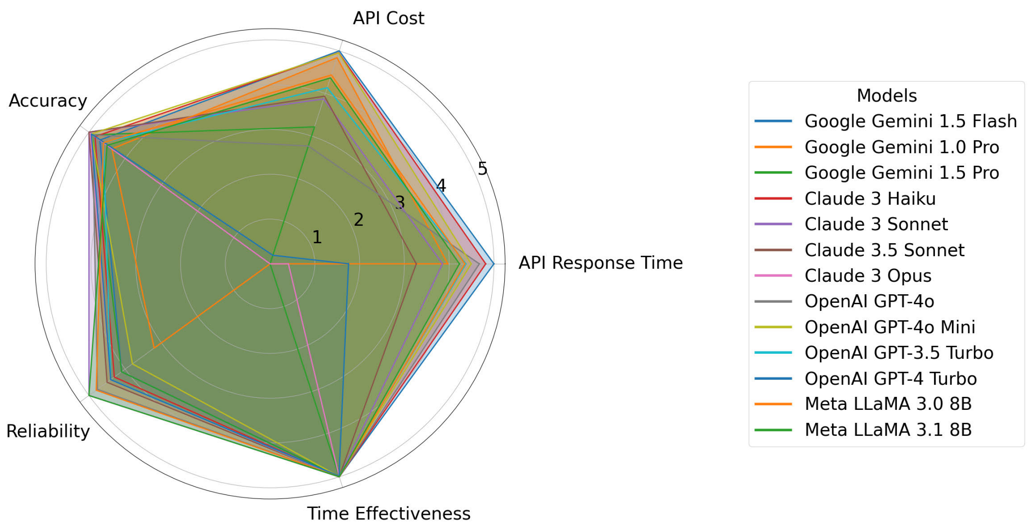

FIGURE 2. Comprehensive Comparison of Model Capabilities with inverted scales, matplotlib librar.

A. COMPARISON OF MODEL PERFORMANCE

Claude 3.5 Sonnet and Claude 3 Sonnet demonstrated strong and consistent performance, as shown in Section IV. Their perfect accuracy scores across both simple and complex tasks indicate their suitability for use in professional development environments where reliability and correctness are key.

From the analysis, it is clear that Anthropic's latest models, such as Claude 3.5 Sonnet, have now surpassed previous leaders like GPT-4, particularly in accuracy and reliability, as shown in [55]. These Claude models consistently rank among the top across all tasks, highlighting their suitability for applications that demand high precision and consistency.

This shift contrasts with earlier evaluations where GPT-4 excelled in various applications, noted for its high-quality code generation capabilities. The measurable advancements in Claude models now position them as a preferred choice for tasks requiring the utmost accuracy and reliability, marking a significant progression in AI-assisted performance.

Conversely, models like Gemini 1.0 Pro exhibited weaker performance, particularly in handling simpler tasks, where lower accuracy scores and inconsistencies in reliability became apparent - even worse than with GPT-4o Mini.

B. IMPACT OF TASK COMPLEXITY

As detailed in the Results section, model performance varied significantly with task complexity. While many models managed simpler problems well, complex tasks such as web application development exposed the limits of models like Gemini 1.0 Pro and GPT-3.5 Turbo. This suggests a need for more context-aware and structurally capable AI models when addressing real-world applications.

C. API COST EFFICIENCY, RESPONSE TIME AND TIME EFFICIENCY

From a practical deployment perspective, API cost and response time are often decisive. Section IV shows that Gemini 1.5 Flash and Claude 3 Haiku provide excellent trade-offs between cost and performance, making them attractive for lightweight or large-scale applications. On the other hand, higher-end models like GPT-4 Turbo and Claude 3 Opus offer better performance, albeit at a premium.

Time efficiency was another key factor in assessing model performance, especially for real-time applications. The results show that while most models achieved similar execution times, the Meta LLaMA models demonstrated shorter execution times, particularly in complex tasks, compared to their competitors. This suggests that these models may be better suited for time-sensitive applications or automated development pipelines.

D. EXCEPTION HANDLING AND CODE COMPLETENESS

Exception handling is essential for generating robust and secure code. As shown in Section IV, only Claude 3.5 Sonnet implemented exception handling consistently without explicit prompting. This suggests a higher contextual understanding and the potential for autonomous mitigation of runtime errors, which is especially valuable in production or security-sensitive environments. This behavior demonstrates a step toward secure-by-design code generation, reducing the risk of unhandled failures in critical systems such as finance, healthcare, or user-facing applications.

Code completeness also varied across models. More complex tasks, such as building a simple web application, revealed gaps in the outputs of several models. This highlights the importance of structural coherence and the ability to manage multi-part solutions — a challenge for some current generative models.

These findings align with prior research showing that even advanced models, while syntactically correct, may struggle with functional completeness and robustness [56]. Variability in output quality, similar to human developers, underscores the continued need for validation and review in AI-supported development workflows [57].

E. IMPLICATIONS FOR AUTOMATED SOFTWARE DEVELOPMENT

The findings of this study offer valuable insights for developers looking to incorporate AI-generated code into their workflows. An important finding is the absence of exception-handling mechanisms in most generated code samples, which could pose security risks if deployed without additional validation. Moreover, while AI-generated code may follow syntactic correctness, the lack of security-oriented coding practices, such as SQL injection prevention and proper input validation, remains a limitation. As AI continues to integrate into automated software development, security-aware code generation should become a focus of future AI model improvements.

Depending on task requirements, developers must carefully balance factors such as accuracy, reliability, execution time, and cost. For instance, while smaller models like Claude 3 Haiku or GPT-4o Mini excel in cost and speed, their lower code quality may limit their suitability for certain projects. This is especially relevant in web application development, where security risks and vulnerabilities are critical concerns, highlighting the need for security-conscious AI models [58].

In contrast, larger models like those in the Claude and GPT series, known for their strong reliability and accuracy, are well-suited for tasks requiring a more complex approach. These models provide a compelling option for developers focused on high-quality, secure code in demanding applications.

F. LIMITATIONS AND FUTURE WORK

This study focused on evaluating generative AI models for code generation using Python. While Python was selected due to its widespread use in AI-assisted software development, the findings may not directly generalize to other programming languages with different structural and syntactic characteristics. As generative AI models continue to evolve, their capabilities may change with updates or new versions. Moreover, this study did not assess the models' performance in generating code for other programming languages or domain-specific applications.

Future research should consider expanding the range of tasks to include other programming languages like Java, C++, or JavaScript.

Extending this evaluation to other programming languages presents several challenges. Unlike Python, languages such as Java and C++ have stricter type systems and require explicit memory management, which may impact the performance of generative models in code synthesis. JavaScript, on the other hand, follows an event-driven paradigm, making AI-generated code evaluation more complex. These differences could influence model accuracy, reliability, and execution efficiency, requiring tailored evaluation criteria. Domain-specific programming tasks (e.g., SQL query generation, optimization-based programming) may also demand alternative assessment methodologies. Addressing these challenges will be crucial for ensuring the broader applicability of AI-assisted code generation.

Additionally, integrating more sophisticated evaluation metrics, such as assessing the optimization level and security of the generated code, could provide deeper insights into each model's practical utility.

While this study evaluates reliability based on consistency across multiple iterations, future work will include quantitative analysis using Self-BLEU scores and embedding-based similarity metrics. These approaches will allow a more precise assessment of response variation and model consistency.

We also need to emphasize the challenges associated with adapting generative models for specific domains, highlighting the need for further research in fine-tuning models for complex tasks [59].

VI. CONCLUSION

This study evaluated several generative AI models for Python code generation across ten varied tasks. Models tested included OpenAI's GPT series, Google's Gemini, Meta's LLaMA, and Anthropic's Claude, with metrics focusing on syntax correctness, code completeness, accuracy, API response time, and cost efficiency.

Results showed clear differences in performance: bigger Claude models (like 3.5 Sonnet, 3 Opus or 3 Sonnet) excelled in accuracy and reliability, making them suitable for high-precision tasks. Conversely, Gemini models struggled with accuracy and reliability, revealing big areas for improvement. Cost-effective models like Claude 3 Haiku and Gemini 1.5 Flash offered budget-friendly solutions with respectable performance on complex tasks.

These insights guide developers on model selection based on accuracy, reliability, and cost-efficiency needs. Compared to existing evaluations of generative AI in code generation, this study enhances benchmarking methodologies by systematically analyzing multi-iteration reliability and incorporating exception handling metrics. This ensures a more robust comparison of model performance in practical coding tasks. Future research could expand to other languages and additional metrics to better assess AI-generated code in practical applications.

This study focuses exclusively on Python, which is widely used in AI-driven software development. However, AI models may perform differently in other programming languages such as Java, C++, or JavaScript, where syntax complexity and execution efficiency vary. Moreover, domain-specific programming tasks (e.g., SQL query generation, optimization-based coding) may require different evaluation approaches. Future research will extend this study by incorporating additional programming languages and task types to better assess the applicability of generative AI in broader software engineering contexts.

Beyond academic significance, the findings offer important implications for real-world adoption. In educational settings, generative AI can support programming instruction through automatic feedback and example generation. In enterprise environments, accurate and consistent models reduce development costs and accelerate delivery. Moreover, models capable of autonomous exception handling provide a foundation for more secure code, which is critical in production and cybersecurity-sensitive applications.

Additionally, future work will incorporate Self-BLEU and embedding-based similarity analysis to measure the diversity and consistency of generated outputs. These advanced evaluation methods will provide deeper insights into the generative capabilities of AI models, especially in handling multiple iterations of the same coding task.

APPENDIX A: MATERIALS ON GITHUB

For the full version of the source code, details about the models used, and more detailed results, please visit the public GitHub repository:

https://github.com/dominikpalla/dpmms (will be updated)

ACKNOWLEDGMENT

The authors would like to express their heartfelt gratitude to Prof. Hana Skalska for her invaluable guidance on the statistical analyses in this article. Her expertise was crucial in shaping the methodological rigor of this work. They also deeply grateful to Prof. Hana Tomaskova for her assistance with the formal aspects of the manuscript, including her advice on methodology and writing structure. Finally, they wish to express their deep appreciation to Prof. Ondrej Krejcar for his practical insights and invaluable support in effectively communicating the results. His guidance significantly enriched the practical relevance of this study. Their combined expertise and support were instrumental to the successful completion of this work. They would also like to acknowledge the valuable assistance provided by generative AI technology, specifically OpenAI's ChatGPT-4 and ChaGPT-4o models. AI-assisted rephrasing played a role in refining the language, enhancing the grammatical accuracy, and improving the overall clarity of the text. This support allowed them to effectively communicate the findings of their research in a more polished and impactful manner [60].

REFERENCES

[1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 30, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., Long Beach, CA, USA. Red Hook, NY, USA: Curran Associates, Dec. 2017, pp. 1–11.
[2] B. Idrisov and T. Schlippe, "Program code generation with generative AIs," *Algorithms*, vol. 17, no. 2, p. 62, Feb. 2024.
[3] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, "Improving language understanding by generative pre-training," OpenAI, Tech. Rep., 2019. Accessed: Apr. 11, 2025. [Online]. Available: https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf
[4] A. R. Sadik, S. Brulin, and M. Olhofer, "Coding by design: GPT-4 empowers agile model driven development," Oct. 2023, *arXiv:2310.04304*.
[5] R. Bommasani et al., "On the opportunities and risks of foundation models," Jul. 2022, *arXiv:2108.0725*.
[6] E. R. Adwaith Krishna, A. Sha, K. Anvesh, N. A. Reddy, B. S. Raj, and K. S. Nisha, "Generative AI-driven approach to converting numerical code into mathematical functions," in *Proc. 2nd Int. Conf. Autom., Comput. Renew. Syst. (ICACRS)*, Dec. 2023, pp. 661–666.
[7] T. Brown et al., "Language models are few-shot learners," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 33, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, Eds., Vancouver, BC, Canada. Red Hook, NY, USA: Curran Associates, Inc., Jul. 2021, pp. 1877–1901.
[8] M. F. Rabbi, A. I. Champa, M. F. Zibran, and M. R. Islam, "AI writes, we analyze: The ChatGPT Python code saga," in *Proc. 21st Int. Conf. Mining Softw. Repositories*, Apr. 2024, pp. 177–181.
[9] D. H. Ackley, G. E. Hinton, and T. J. Sejnowski, "A learning algorithm for Boltzmann machines," *Cognit. Sci.*, vol. 9, no. 1, pp. 147–169, Mar. 1985.
[10] A. Fan, M. Lewis, and Y. Dauphin, "Hierarchical neural story generation," in *Proc. 56th Annu. Meeting Assoc. Comput. Linguistics*, I. Gurevych and Y. Miyao, Eds., Melbourne, VIC, Australia: Association for Computational Linguistics, Jun. 2018, pp. 889–898.
[11] A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi, "The curious case of neural text degeneration," in *Proc. 8th Int. Conf. Learn. Represent. (ICLR)*, Addis Ababa, Ethiopia, Apr. 2020, pp. 1–16. Accessed: Apr. 11, 2025. [Online]. Available: https://arxiv.org/pdf/1904.09751.pdf
[12] Aarti, "Generative AI in software development: An overview and evaluation of modern coding tools," *Int. J. For Multidisciplinary Res.*, vol. 6, no. 3, pp. 1–9, May/Jun. 2024. Accessed: Apr. 11, 2025. [Online]. Available: https://www.ijfmr.com/papers/2024/3/23271.pdf
[13] 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*. Cham, Switzerland: Springer, 2024, pp. 151–172, doi: 10.1007/978-3-031-55642-5_7.
[14] M. Acher, "A demonstration of end-user code customization using generative AI," in *Proc. 18th Int. Work. Conf. Variability Model. Software-Intensive Syst.*, Bern, Switzerland, Feb. 2024, pp. 1–6.
[15] A. Pilipiszyn. *GPT-3 Powers the Next Generation of Apps*. Accessed: Feb. 27, 2025. [Online]. Available: https://openai.com/index/gpt-3-apps/
[16] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *J. Mach. Learn. Res.*, vol. 21, no. 140, pp. 1–67, 2020. Accessed: Apr. 11, 2025. [Online]. Available: https://www.jmlr.org/papers/volume21/20-074/20-074.pdf
[17] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics, Human Lang. Technol.*, Minneapolis, MN, USA, J. Burstein, C. Doran, and T. Solorio, Eds., Jan. 2018, pp. 4171–4186.
[18] S. S. Kumar, M. A. Lones, M. Maarek, and H. Zantout, "Investigating the proficiency of large language models in formative feedback generation for student programmers," in *Proc. 1st Int. Workshop Large Lang. Models Code*. New York, NY, USA: Association for Computing Machinery, Apr. 2024, pp. 88–93.
[19] B. A. Becker, P. Denny, J. Finnie-Ansley, A. Luxton-Reilly, J. Prather, and E. A. Santos, "Programming is hard—Or at least it used to be: Educational opportunities and challenges of AI code generation," in *Proc. 54th ACM Tech. Symp. Comput. Sci. Educ.* New York, NY, USA: Association for Computing Machinery, Mar. 2023, pp. 500–506.
[20] S. Bhatia, T. Gandhi, D. Kumar, and P. Jalote, "Unit test generation using generative AI: A comparative performance analysis of autogeneration tools," in *Proc. 1st Int. Workshop Large Lang. Models Code*. New York, NY, USA: Association for Computing Machinery, Apr. 2024, pp. 54–61.
[21] J. D. Weisz, M. Müller, S. I. Ross, F. Martinez, S. Houde, M. Agarwal, K. Talamadupula, and J. T. Richards, "Better together? An evaluation of AI-supported code translation," in *Proc. 27th Int. Conf. Intell. User Interfaces*. New York, NY, USA: Association for Computing Machinery, Mar. 2022, pp. 369–391.

[22] P. Valero-Lara, A. Huante, M. A. Lail, W. F. Godoy, K. Teranishi, P. Balaprakash, and J. S. Vetter, ''Comparing Llama-2 and GPT-3 LLMs for HPC kernels generation,'' Sep. 2023, *arXiv:2309.07103*.
[23] C. Tony, N. E. D. Ferreyra, M. Mutas, S. Dhiff, and R. Scandariato, ''Prompting techniques for secure code generation: A systematic investigation,'' 2024, *arXiv:2407.07064*.
[24] GitHub. (2024). *GitHub Copilot · Your AI Pair Programmer*. [Online]. Available: https://github.com/features/copilot
[25] Amazon. *Code Review Tool—Amazon CodeGuru Security—AWS*. Accessed: Feb. 27, 2025. [Online]. Available: https://aws.amazon.com/codeguru/
[26] OpenAI. *OpenAI Platform*. Accessed: Feb. 27, 2025. [Online]. Available: https://platform.openai.com
[27] OpenAI. *Introducing GPT-4o and More Tools to ChatGPT Free Users*. Accessed: Feb. 27, 2025. [Online]. Available: https://openai.com/index/gpt-4o-and-more-tools-to-chatgpt-free/
[28] OpenAI. *GPT-4o Mini: Advancing Cost-Efficient Intelligence*. Accessed: Feb. 27, 2025. [Online]. Available: https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/
[29] J. Kemper. (Mar. 2023). *OpenAI Kills Its Codex Code Model, Recommends GPT3.5 Instead*. [Online]. Available: https://the-decoder.com/openai-kills-code-model-codex/
[30] Google. (Sep. 2023). *Bard Can Now Connect to Your Google Apps and Services*. [Online]. Available: https://blog.google/products/gemini/google-bard-new-features-update-Sep.-2023/
[31] Google. (Oct. 2024). *Gemini—Google DeepMind*. [Online]. Available: https://deepmind.google/technologies/gemini/
[32] Google. *Introducing Gemini: Google's Most Capable AI Model Yet*. Accessed: Feb. 27, 2025. [Online]. Available: https://blog.google/technology/ai/google-gemini-ai/#sundar-note
[33] Google. *Gemini Models | Gemini API*. Accessed: Feb. 27, 2025. [Online]. Available: https://ai.google.dev/gemini-api/docs/models/gemini
[34] Google. *Introducing Gemini 1.5, Google's Next-generation AI Model*. [Online]. Available: https://blog.google/technology/ai/google-gemini-next-generation-model-february-2024/#sundar-note
[35] (Nov. 2024). *Meta-Llama/Llama3, Meta Llama*. Accessed: Feb. 27, 2025. [Online]. Available: https://huggingface.co/meta-llama/Meta-Llama-3-8B
[36] Meta. (Sep. 2024). *Meta-llama/Llama-3.1-8B · Hugging Face*. [Online]. Available: https://huggingface.co/meta-llama/Llama-3.1-8B
[37] *Introducing Llama 3.1: Our Most Capable Models to Date*. Accessed: Feb. 27, 2025. [Online]. Available: https://ai.meta.com/blog/meta-llama-3-1/
[38] Anthropic. *Introducing Computer Use, a New Claude 3.5 Sonnet, and Claude 3.5 Haiku*. Accessed: Feb. 27, 2025. [Online]. Available: https://www.anthropic.com/news/3-5-models-and-computer-use
[39] *Claude 3 Models Compared: Opus, Sonnet, Haiku | Claude AI Hub*, Anthropic, San Francis, CA, USA, Apr. 2024.
[40] Anthropic. *Claude 3 Haiku: Our Fastest Model Yet*. Accessed: Feb. 27, 2025. [Online]. Available: https://www.anthropic.com/news/claude-3-haiku
[41] JetBrains. *Code Inspections | PyCharm Documentation*. Accessed: Feb. 27, 2025. [Online]. Available: https://www.jetbrains.com/help/pycharm/code-inspection.html
[42] M. Chen et al., ''Evaluating large language models trained on code,'' 2021, *arXiv:2107.03374*.
[43] 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, May 2024.
[44] V. Adamson, ''Assessing the effectiveness of ChatGPT in generating Python code,'' Ph.D. dissertation, School Inform., Univ. Skövde, Skövde, Sweden, 2023.
[45] R. Arias, G. Martinez, D. Cáceres, and E. Garces, ''Limitations and benefits of the ChatGPT for Python programmers and its tools for evaluation,'' in *Cybernetics and Control Theory in Systems*, R. Silhavy and P. Silhavy, Eds., Cham, Switzerland: Springer, 2024, pp. 171–194.
[46] B. Yetiştiren, I. Özsoy, M. Ayerdem, and E. Tüzün, ''Evaluating the code quality of AI-assisted code generation tools: An empirical study on GitHub copilot, Amazon CodeWhisperer, and ChatGPT,'' Oct. 2023, *arXiv:2304.10778*.
[47] C. E. A. Coello, M. N. Alimam, and R. Kouatly, ''Effectiveness of ChatGPT in coding: A comparative analysis of popular large language models,'' *Digital*, vol. 4, no. 1, pp. 114–125, Jan. 2024.
[48] Anthropic. *Welcome to Claude*. Accessed: Feb. 27, 2025. [Online]. Available: https://huggingface.co/meta-llama/Llama-3.1-8B
[49] IntelliJ. *PyCharm: The Python IDE for Data Science and Web Development*. Accessed: Feb. 27, 2025. [Online]. Available: https://huggingface.co/meta-llama/Llama-3-8B
[50] Graffatcolmingov, Lukasa, and Nateprewitt, *Requests: Python HTTP for Humans*, Python Software Found., Beaverton, OR, USA, 2023.
[51] Porter97, *Pandas3: Boto3 Extension to Help Facilitate Data Science Workflows With S3 and Pandas*, Independent GitHub Developers, GitHub, Inc., San Francisco, CA, USA, 2024.
[52] Ivanov, Matthew. Brett, and Mdboom2, *Matplotlib: Python Plotting Package*, Matplotlib Develop. Team, NumFOCUS, Austin, TX, USA, 2024.
[53] Hauntsaninja, *Tiktoken: Tiktoken is a Fast BPE Tokeniser for Use With OpenAI's Models*, OpenAI, San Francisco, CA, USA, 2024.
[54] Amysartran, Arthurzucker, lysandre, and Thomwolf, *Transformers: State-of-the-Art Machine Learning for JAX, PyTorch and TensorFlow*, Hugging Face, Inc., New York, NY, USA, 2024.
[55] T. Phung, V.-A. Pădurean, J. Cambronero, S. Gulwani, T. Kohn, R. Majumdar, A. Singla, and G. Soares, ''Generative AI for programming education: Benchmarking ChatGPT, GPT-4, and human tutors,'' in *Proc. ACM Conf. Int. Comput. Educ. Res.* New York, NY, USA: Association for Computing Machinery, Aug. 2023, pp. 41–42.
[56] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, and C. Sutton, ''Program synthesis with large language models,'' Aug. 2021, *arXiv:2108.07732*.
[57] A. Clark, D. Igbokwe, S. Ross, and M. F. Zibran, ''A quantitative analysis of quality and consistency in AI-generated code,'' in *Proc. 7th Int. Conf. Softw. Syst. Eng. (ICoSSE)*, Apr. 2024, pp. 37–41.
[58] B. Zhu, N. Mu, J. Jiao, and D. Wagner, ''Generative AI security: Challenges and countermeasures,'' Oct. 2024, *arXiv:2402.12617*.
[59] X. Gu, M. Chen, Y. Lin, Y. Hu, H. Zhang, C. Wan, Z. Wei, Y. Xu, and J. Wang, ''On the effectiveness of large language models in domain-specific code generation,'' *ACM Trans. Softw. Eng. Methodol.*, vol. 34, no. 3, pp. 1–22, Mar. 2025.
[60] J. Achiam et al., ''GPT-4 technical report,'' Mar. 2024, *arXiv:2303.08774*.

DOMINIK PALLA received the M.Sc. degree in applied informatics from the Faculty of Informatics and Management, University of Hradec Kralove, Czech Republic, where he is currently pursuing the Ph.D. degree in applied informatics.

Since 2020, he has been involved in various research projects, including international collaborations, such as COST LITHME and DX.SEA. He is currently a Lecturer in software development and algorithmization with the Department of Informatics and Quantitative Methods, Faculty of Informatics and Management, University of Hradec Kralove. His work includes contributions to project design and implementation in the areas of AI and software automation. He is also a Freelance AI Consultant and has a lot of experience with entrepreneurship. His research interests include generative artificial intelligence in software development, generative AI, automated software development, and interdisciplinary applications of AI in informatics.

ANTONIN SLABY received the full professorship in mathematics, with a focus on computer science from the University of Life Sciences, Prague, Czech Republic. He is currently with the Department of Informatics and Quantitative Methods, Faculty of Informatics and Management, University of Hradec Kralove, where he has significantly contributed to the field through numerous research projects. He held several key positions with the University of Hradec Kralove, including the Vice-Rector and the Head of the Department of Informatics and Quantitative Methods. He has also participated in various international projects, focusing on advanced mathematical applications in computer science. His teaching covers algorithmization and discrete mathematics. He has authored multiple research articles in these fields.

Prof. Slaby is a member of several professional boards and organizations and is commonly honored for his contributions to academic leadership and research. He continues to play an active role in lecturing Ph.D. students.