EVALUATING AI-GENERATED CODE FOR C++, FORTRAN, GO, JAVA, JULIA, MATLAB, PYTHON, R, AND RUST

A PREPRINT

Patrick Diehl
Louisiana State University
Baton Rouge, 70803, LA, USA
Applied Computer Science,
Los Alamos National Laboratory
pdiehl@cct.lsu.edu

Noujoud Nader
Louisiana State University
Baton Rouge, 70803, LA, USA
nnader@lsu.edu

Steve Brandt
Louisiana State University
Baton Rouge, 70803, LA, USA
sbrandt@@cct.lsu.edu

Hartmut Kaiser
Louisiana State University
Baton Rouge, 70803, LA, USA
hkaiser@@cct.lsu.edu

ABSTRACT

This study evaluates the capabilities of ChatGPT versions 3.5 and 4 in generating code across a diverse range of programming languages. Our objective is to assess the effectiveness of these AI models for generating scientific programs. To this end, we asked ChatGPT to generate three distinct codes: a simple numerical integration, a conjugate gradient solver, and a parallel 1D stencil-based heat equation solver. The focus of our analysis was on the compilation, runtime performance, and accuracy of the codes. While both versions of ChatGPT successfully created codes that compiled and ran (with some help), some languages were easier for the AI to use than others (possibly because of the size of the training sets used). Parallel codes—even the simple example we chose to study here—are also difficult for the AI to generate correctly.

Keywords AI-generated code · C++ · Julia · Java · Matlab · Julia · Python · R · Rust

1 Introduction

Generative AI algorithms have been used to create codes and high-quality content quickly and cost-effectively, tailored to specific user needs or criteria [20, 19]. This new tool has the potential to revolutionize traditional programming methodologies and change the way code is developed [3]. GitHub Copilot [5], for instance, is enhancing the developer experience by introducing features like Copilot Chat [11]. AI code generation tools typically operate on a tiered subscription model, with monthly per-user prices ranging from $10 to $25 at the time of this writing. Many providers also offer a free version tailored for individual users. The lower-priced tiers generally provide only basic functionality such as manual code generation through prompts and autocompletion. More advanced features, including model tuning and security scanning, are typically reserved for the more expensive tiers [7].

As these tools become more integrated into everyday coding practices, they pose both opportunities and challenges that merit careful examination and ongoing research. Recent advancements in AI-generated code represent a continuation of the challenges facing traditional computer science education. The emergence of AI code generators, coupled with the abundant availability of online code and platforms that enable contract cheating, starkly contrasts with the traditional image of programmers diligently crafting bespoke code [2]. In addition, while these tools offer the potential to simplify the coding process and make programming more accessible to beginners, it is essential to recognize their limitations. These tools may not consistently generate code that is correct, efficient, or easy to maintain. Consequently, it is critical to thoroughly evaluate multiple factors when considering the adoption of such tools for coding tasks [21].

In November 2022, OpenAI introduced ChatGPT, a derivative of the generative pre-trained transformer (GPT), a large language model (LLM) based on transformer architecture that can comprehend human languages and generate text resembling human writing. Within just five days after its launch, the platform had already registered over one million users [16]. OpenAI improved the program on March 14th, 2023, with the release of GPT 4, which promised better reasoning ability. In this study, we systematically evaluate the accuracy and efficacy of the code generated by ChatGPT both versions 3.5 and 4 in many programming languages. Our analysis focuses on the compilation and correctness of the generated code. Through this study, we aim to establish a comprehensive understanding of the AI-code generator ChatGPT's capabilities and its limitations. We selected programming languages from among the Top 20 in the TIOBE Index [18]. The languages chosen for our analysis include C++, Fortran, Go, Java, Julia, Matlab, Python, R, and Rust. The first example in this study is an example of numerical integration; the second, a conjugate gradient solver that incorporates matrix and vector operations; and lastly, a parallel 1D stencil-based heat equation solver [6]. This last model problem was chosen due to its relevance in testing parallel computational workflows and its common usage in scientific computing.

The paper is structured as follows: In Section 2, we will provide an overview of related work. In Section 3, we will present our methodology. The quality of the generated code and code metrics will be described in Sections 4 and 5. In Section 6, we conclude our work and indicate possible future steps.

2 Related Work

Previous studies evaluated the performance of AI models for code generation using different strategies. Chen at al. [4] released HumanEval, a new evaluation set measuring functional correctness for synthesizing programs from docstrings. They find that HumanEval solves 28.8% of the problems while GPT-3 solves 0%. Nguyen et al. in [10] conducted an empirical investigation to assess the correctness and understandabilty of Copilot's proposed code. They found that Copilot's suggestions for Java had the highest correctness rate at 57%, whereas JavaScript had the lowest correctness rate at 27%. Another study evaluated the performance of an AI tool named Codex in localizing and fixing bugs [12, 13]. Instead of generating code, Tang et al. used eye tracking and IDE tracking [17] to conduct a user study to assess how programmers deal with errors when using Copilot. Recently, Liu et al. [8] systematically studied the quality of $4,066$ ChatGPT-generated code implemented in Java and Python, for $2,033$ programming tasks. They first analyzed the correctness of ChatGPT when generating code and found that while it could generate functional code, there were a number of quality errors. Providing the AI with feedback and allowing it to self-repair was of limited effectiveness in removing these quality issues.

These studies, we think, show the potential and promise for AI code generation, but also highlight its present weaknesses.

3 Methodology

We use three numerical examples: numerical integration (**NI**), a conjugate gradient solver (**CGS**), and a parallel one-dimensional heat equation solver (**PHS**) using finite differencing. The code complexity increases with each example. We used the free version of ChatGPT 3.5 and the paid version ChatGPT 4.0 for our study. We used ChatGPT to generate the codes on *02/27/2024* for the last example and *06/05/2024* for the others. The following queries were used to obtain the source code;

1. Write a **language** code to compute the area between $-\pi$ and $2/3\pi$ for $\sin(x)$ and validate it.
   Here, we want to validate if ChatGPT can write a code to evaluate

$$\int\limits_{-\pi}^{2/3\pi} \sin(x)dx. \tag{1}$$

2. Write a conjugate gradient solver in **language** to solve A times x equals b and validate it.
   Here, we want to evaluate whether ChatGPT can write a conjugate gradient solver and apply it to a linear system of equations, i.e.

$$A^{n\times n} \cdot x^n = b^n \text{ with } n \in \mathbb{Z}^+, A = A^T, \text{and } x^T Ax > 0, \text{for all } X \in \mathbb{R}^n. \tag{2}$$

   For more details about the conjugate gradient solver, we refer to [14].

3. Write a parallel 1D heat equation solver using finite differencing in **language**.
   Here, we want to evaluate whether ChatGPT can write the code to solve

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2}, \quad 0 \le x < L, t > 0 \tag{3}$$

Table 1: Three aspects of code quality: Compilation (Did the code compile with a recent compiler?), Runtime (Did the code execute without errors?), and Correctness (Did the code compute correct results?).

Attribute	V	C++	Python	Julia	Java	Go	Rust	Fortran	Matlab	R
*Numerical integration (**NI**)*										
Compila-tion	3.5	✓	–	–	✓	✓	✓	✗	–	–
	4.0	✓	–	–	✓	✓	✓	✓	–	–
Runtime	3.5	✓	✓	✓	✓	✓	✓	✓	✓	✓
	4.0	✓	✗	✓	✓	✓	✓	✓	✓	✓
Correct-ness	3.5	✓	✓	✓	✓	✓	✓	✓	✓	✓
	4.0	✓	✗	✗	✗	✗	✗	✗	✗	✗
*Conjugate gradient solver (**CGS**)*										
Compila-tion	3.5	✓	–	–	✓	✓	✗	✗	–	–
	4.0	✓	–	–	✓	✓	✓	✓	–	–
Runtime	3.5	✓	✓	✓	✓	✓	✓	✓	✓	✓
	4.0	✓	✓	✗	✓	✓	✓	✓	✓	✗
Correct-ness	3.5	✓	✓	✓	✓	✗	✓	✓	✓	✓
	4.0	✓	✓	✓	✓	✓	✓	✓	✓	✗

where α is the material's diffusivity. For the discretization in space a finite difference scheme

$$u(x_i, t+1) = u(x_i, t) + dt \, \alpha \frac{u(t, x_{i-1}) - 2u(t, x_i) + u(t, x_{i+1})}{2h} \tag{4}$$

We did not specify the how to generate the grid, *i.e.* equidistant nodal spacing with n grid points $x = \{x_i = i \cdot h \in \mathbb{R} | i = 0, \ldots, n-1\}$, nor what time integration method to use, *e.g.* the Euler method.

Whether the language was C++, Fortran, Go, Java, Matlab, Python, R, and Rust, we copied the generated code to the paper's GitHub repo[1]. For some of the generated codes, ChaptGPT provided instructions on how to compile the code. This instruction was added to the *Readme.md*.

4 Quality of the generated software

In this section, we assess the quality of the AI-generated software. First, we checked whether the code compiles with a recent compiler. Second, we checked whether the code executed without segmentation faults or other runtime errors. Third, we checked whether the code produced a correct result. The results for our test cases were the following:

$$\int_{-\pi}^{2/3\pi} \sin(x) dx = -\cos(2/3\pi) + cos(-\pi) = -0.5$$

We solve $M \times x = b$ with the following values $A = \begin{pmatrix} 4 & 1 \\ 1 & 3 \end{pmatrix}$ and $b = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$. The solution x is $(0.09090909 \quad 0.63636364)^T$.

Table 1 summarizes the results for the first two examples (NI and CGS). All codes and hand-repaired codes are available on GitHub[1]. However, for the sake of space we can not show them in the paper. We added the compilation and runtime errors to GitHub as well. The results are shown in Table 2. Table 3 lists the versions of the tools used.

C++ The trapezoidal rule was used for the numerical integration. Both codes finished and returned the correct result. The ChaptGPT 4.0 code computed two results once using the trapezoidal rule, with and without absolute values of the function. For the conjugate gradient solver both codes compiled, finished, and produced correct results. For, the parallel heat equation solver, OpenMP was used for parallelism. The code generated by ChatGPT

---
[1]https://github.com/diehlpkpapers/heat-ai/tree/main

Table 2: Three aspects of code quality for the **parallel heat equation solver (PHS)**: Compilation (Did the code compile with a recent compiler?), Runtime (Did the code execute without errors?), and Correctness (Did the code compute correct results?).

Attribute	V	C++	Kokkos	HPX	Python	Julia	Java	Go	Rust	Fortran	Matlab	R
Compila-tion	3.5	✓	✓	✓	–	–	✓	✓	✗	✗	–	–
	4.0	✗	✓	✓	–	–	✓	✓	✓	✓	–	–
Runtime	3.5	✓	✓	✗	✗	✓	✗	✗		✓	✓	✗
	4.0	✓	✓	✓	✗	✗	✓	✗	✓	✓	✓	✓
Correct-ness	3.5	✓	✗	✓	✗	✗	✓	✗		✗	✓	✓
	4.0	✓	✗	✗	✓	✓	✓	✗	✗	✗	✓	✓

Table 3: Version of all used tools in this study

Tool	C++	Python	Julia	Java	Go	Rust	Fortran	Matlab	R	Kokkos	HPX
Version	12	3.9.18	1.10.3	22	1.20.12	1.70	12	R2024a	4.4.0		1.9.1

3.5 compiled, the code generated by ChatGPT 4.0 had issues with shared variables in the OpenMP parallel region (`error: r is predetermined shared` **for** shared). Both codes had no runtime errors and produced plausible results.

Kokkos The ChatGPT 3.5 version and ChatGPT 4.0 version used a Kokkos:: `parallel_for` loop for parallel computing. Both codes were compiled with the latest Kokkos version using `CMake`. Both codes produced incorrect results.

HPX The generated code for ChatGPT 3.5 included some HPX-specific headers but none of the parallel constructs. The code generated by ChatGPT 4.0 used HPX's parallel algorithms for parallelism. Both codes compiled after small changes in header names and namespaces due to the HPX version. ChatGPT did not mention which HPX version to use. All values for the ChatGPT 4 version produced NaN.

Python In the NI example, the code generated by ChatGPT 3.5 ran without runtime errors and produced accurate results. ChatGPT 4.0 provided two codes, one utilizing the numpy library and the other employing the scipy library. Upon testing both, a runtime error occurred in the numpy-based code due to a `TypeError: bad operand type`. Unfortunately, neither code option from ChatGPT 4.0 produced correct results. In the CGS example, both code snippets ran without runtime errors and yielded correct results. In the PHS example, the code generated by ChatGPT 3.5 utilized the *joblib* library to implement parallelism, whereas the code from ChatGPT 4.0 used the multiprocessing library. Although both pieces of code were syntactically correct and valid Python code, they each encountered runtime errors. However, while the errors in the ChatGPT 3.5-generated code remain unresolved, the issues in the ChatGPT 4.0-generated code were promptly addressed, and provided correct results.

Julia For the numerical integration, the ChatGPT 4.0 version computed 3.5 for the integral by using the absolute value of the function for the trapezoidal rule. For the conjugate gradient solver, the ChatGPT 4.0 code stopped with `ERROR: LoadError: UndefVarError: 'dot' not defined`. After copying the dot product function from the ChatGPT 3.5 version, the code worked. Both codes returned the correct result. For the parallel heat equation solver, the ChatGPT 3.5 version produced final values were zero (which was incorrect). A short time later, the code stopped with the error `Unhandled Task ERROR: On worker 2: UndefVarError: 'u' not defined`. The ChatGPT 4.0 code did not print anything and crashed with `KeyError: key SharedArrays [1a1011a3-84de-559e-8e89-a11a2f7dc383] not found`. The ChatGPT 3.5 version used a for loop for parallelism (`@distributed`) and ChaptGPT 4 used `@spawnat` for parallelism. The ChatGPT 3.5 version incorrectly parallelized the loop for the time steps instead of the loop for the domain. The ChatGPT 4.0 version added code to plot the results, however, we changed the code to print the results.

Java For the NI example, both codes successfully compiled and ran. The code generated by ChatGPT 3.5 produced accurate results, while the code generated by ChatGPT 4.0 did not. In the CGS example, both codes successfully compiled, ran, and produced correct results. In the PHS example, both the generated codes used Java's built-in

4

concurrency utilities, particularly the ExecutorService from the java . util . concurrent package. This utility manages a pool of threads and allows user to execute tasks in parallel. Both codes compiled, but the ChatGPT 3.5-generated code encountered runtime errors. This error was Index Out Of Bounds Exception which we fixed. The code generated by ChatGPT 4.0 had no errors.

Go For the numerical integration, ChatGPT 4.0 computed the integral as 3.5 while using absolute values for the function evaluation. Everything else was fine. For the conjugate gradient solver all codes worked. For the parallel heat equation, the ChatGPT 3.5 version of the code compiled after fixing the error "math" imported and not used by removing the unused package. The ChatGPT 4.0 compiled without issue. The ChatGPT 3.5 code crashed with the error panic : runtime error : index out of range [−1] with length 101. After fixing the the code by hand, it crashed with the error panic : runtime error : index out of range [101] with length 101. The final values were zero or one and the result was not correct. The ChatGPT 4.0 code crashed with the error panic : sync : negative WaitGroup counter. After fixing the error most of the final values were not-a-number (NaN).

Rust For the numerical integration, the ChatGPT 4.0 version evaluated the integral as 3.5. Everything else worked for both codes. For the conjugate gradient solver, the ChatGPT 3.5 version did not compile because extern **crate** special ; was missing at the top of the main.rs. After fixing that problem, both codes executed and provided the correct result. For the parallel heat equation solver, both codes used the data-parallelism library Rayon for parallelism. The code for the ChatGPT 4.0 version compiled, however, extern **crate** rayon; was missing. However, some of the values were negative which is not correct.

Fortran For the numerical integration, the ChatGPT 3.5 code did not compile due to Error : EXTERNAL attribute conflicts with FUNCTION attribute. The ChatGPT 4.0 code computed the integral as 3.5 by using absolute values for the function evaluation for the trapezoidal rule. For the conjugate gradient solver, the ChatGPT 3.5 version did not compile since there was a variable **dot_product** and a function **dot_product** defined. After fixing this error by hand, both codes compiled and provided correct results. For the parallel heat equation solver, OpenMP was used by ChaptGPT 3.5, and Fortran coarray by ChatGPT 4.0. Neither of these codes compiled. The errors were Error : Symbol t at (1) already has basic **type** of REAL and Error : !$OMP PARALLEL DO iteration variable must be of **type** integer at (1). After fixing these compilation errors by hand, we had to install Fortran coarray. Neither code produced correct results.

Matlab In the NI example, both codes ran without runtime errors. The code generated by ChatGPT 3.5 yielded accurate results, whereas the code generated by ChatGPT 4.0 failed to produce correct results. For the conjugate gradient solver, both codes ran without errors and produced accurate results. In the PHS example, the code generated by ChatGPT 3.5 run without errors, but it did not utilize parallel computing. In contrast, the code generated by ChatGPT 4.0 effectively used the Parallel Computing Toolbox, ran without errors, and produced correct results.

R As presented in table 1, both codes ran without runtime errors for the numerical integration. The code generated by ChatGPT 3.5 produced accurate results, while the code generated by ChatGPT 4.0 did not produce correct results. In the CGS example, the ChatGPT 3.5-code ran without errors and produced correct results. However, runtime errors occurs with the ChatGPT 4.0-generated code, and these remain unaddressed. In the PHS example, the ChatGPT 3.5-generated code used loops without parallelism since the loops were relatively small and the computations were not too expensive. The originally-generated code encountered an error related to a missing package ('*animation*'). To fix this error, we used a different approach and plotted the output, the results were accurate. The code generated with ChatGPT 4 used the *parallel* library and produced correct results.

4.1 Common issues

Here we present the list of common issues which we observed while debugging:

- **Compilation**:
  For NI and CGS, most compilation errors were minor and could be easily fixed. For PHS, most of the errors were related to parallelism, and sometimes knowledge about the parallel programming language or library was needed to fix the problem. In total, we had five compilation errors for all examples.
- **Runtime**:
  Most of the runtime errors were minor and could be easily fixed. Most were type errors, undefined variables, and undefined functions for interpreted languages. Other errors were index out-of-bound exceptions for the
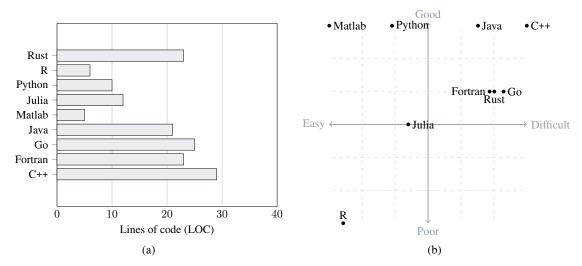
Figure 1: Software engineering metrics for the **numerical integration example**: (a) Lines of code for all implementations using the maximal lines of code. The numbers were determined with the Linux tool *cloc* and (b) Two-dimensional classification using the computational time and the COCOMO model.

heat equation solver for the first and last element with the stencil. Some errors were related to the parallelism and knowledge about the parallel programming language or library was required to address them.

- **Correctness**:
  For NI, the ChatGPT 3.5 versions produced all the correct results. The ChatGPT 4.0 version computed the integral as 3.5 because it used absolute values for the function evaluation. After removing the absolute values all codes computed the correct result. For the conjugate gradient solver, all except two codes produced the correct result. For the single-threaded codes, overall most results were correct. For the parallel codes, 11 codes produced correct results, and 10 codes did not.

5 Code metrics

The lines of code were determined with the Linux tool `cloc` and the larger amount of code lines between the ChaptGPT 3.5 and ChatGPT 4.0 were chosen. The difference was between one and five lines of code. The lines of code were in the same ballpark for all of the codes. The lines of code metric, however, does not measure quality well. For this, we use the **Co**nstructive **Co**st **Mo**del (COCOMO) [15, 1]. COCOMO is a general model with no specialization for parallel programming. There were some attempts to add this capability for the COCOMO II model [9]. Until we get a specialized model for parallel computing or HPC, the COCOMO model is a reasonable candidate for measuring quality. We use the tool *Sloc Cloc and Code* (scc)[2] to get the COCOMO metrics for each approach. We use the average of the COCOMO metric for version 3.5 and version 4.0 for the classification of **easy** and **difficult**. We use the three quantities of interest in Table 1 for the code quality. We use the following metric

$$q(\text{language}) := \frac{1}{2} \left(\frac{\text{comp} + \text{run} + \text{correct}}{3} + \frac{\text{comp} + \text{run} + \text{correct}}{3} \right) \tag{5}$$

to classify the code quality from **poor** to **good**. Figure 1a shows the lines of codes for the numerical integration. Here, Matlab and R have the least lines of code. Python and Julia are comparable. Java, Go, Fortran, and C++ have comparable results. Figure 1b shows the two-dimensional classification. Concerning the code quality, Matlab, Python, Java, and C++ showed the best results. Fortran, Go, and Rust, are very close for code quality and complexity. Julia and R are last. Figure 2a shows the lines of code for the conjugate gradient method. Here, R, Fortran, Julia, and Matlab have comparable results. Fortran and Rust are close. Java, Go, Fortran, and C++ are comparable. Figure 2b shows the two-dimensional classification. Here, Go, Java, and C++ are on the upper end for the code quality. Fortran and Rust are in the middle for code quality and on the opposite side for complexity. Julia and Matlab are second last and Python and R are last. Figure 3a shows the lines for the parallel heat equation solver. Here, Kokkos, Java, and Go are at the top.

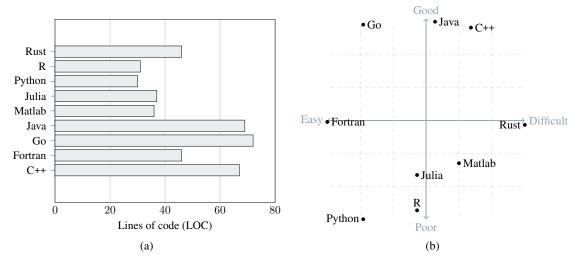---

[2]https://github.com/boyter/scc

Figure 2: Software engineering metrics for the **conjugate gradient solver**: (a) Lines of code for all implementations using the maximal lines of code. The numbers were determined with the Linux tool *cloc* and (b) Two-dimensional classification using the computational time and the COCOMO model.
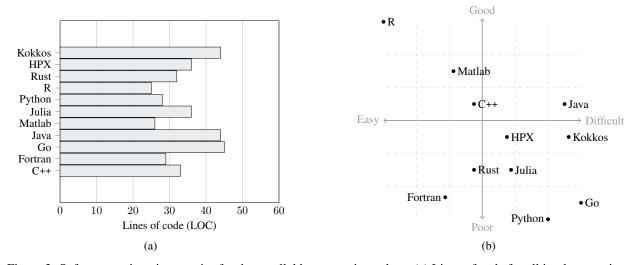

Figure 3: Software engineering metrics for the parallel heat equation solver: (a) Lines of code for all implementations using the maximal lines of code. The numbers were determined with the Linux tool *cloc* and (b) Two-dimensional classification using the computational time and the COCOMO model.

HPX, Python, Rust, C++, and Julia are in the middle. R, Matlab, and Fortran are at the lower end. Figure 3b shows the classification of the parallel heat equation solver. Matlab resulted in the best code quality. Next, R, C++, and Java had a similar code quality. Looking at GitHub C++, Python, and Java were in the top ten most used languages in 2022. Maybe the larger training data set explains why C++ and Java had good results. However, Python is an outlier. It is the second most used language on GitHub, but the code quality was low.

6 Discussion and Conclusion

In this work we have conducted an evaluation of three computational problems using ChatGPT versions 3.5 and 4.0 for code generation using a range of programming languages. We evaluated the compilation, runtime errors, and accuracy of the codes that were produced. We tested their accuracy, first with a basic numerical integration, the with a conjugate gradient solver, and finally with a 1D stencil-based heat equation solver.

7

For the numerical integration example, codes generated by both versions compiled successfully in all languages except Fortran, and executed without any runtime errors. However, the accuracy of the outputs from the ChatGPT 4.0-generated codes was incorrect, possibly due to the misinterpretation of the keyword "area" in the prompt. In the case of the conjugate gradient solver, all generated codes compiled successfully with the exceptions of Fortran and Rust. Despite these compilation issues, the resultant codes from all other languages produced correct results, except for R. The parallel 1D heat problem proved to be the most challenging for the AI. Compilation errors were noted in the codes for Fortran, Rust, and C++. Furthermore, a majority of the generated codes encountered runtime errors, and most failed to produce correct results, indicating substantial issues with the implementation logic or the handling of parallel computing constructs by the AI code generator models.

We then analyzed the lines of code for all the generated codes, and the code quality using the COCOMO metric. The analysis of lines of code across all examples showed that Matlab and R consistently produced the lower lines of codes values, followed by Python, Julia, and Fortran (Section 5). In terms of code quality, C++ and Java consistently demonstrated robustness across all the examples tested, followed by Matlab. These languages appear to offer a balance between code quality and complexity, making them suitable choices for more complex computational tasks.

For future work, we plan to study whether ChaptGPT can be made to fix its code bugs by submitting new requests. Another follow-up would be to study how well ChatGPT can produce code for native GPU kernels. Note that the Kokkos codes we have generated can already run on the GPU. Another direction for study would be to evaluate the performance of the AI-generated code. Lastly, prompting it to create distributed code versions would be interesting.

Acknowledgments

References

[1]   Boehm Barry et al. "Software engineering economics". In: *New York* 197 (1981).

[2]   Brett A Becker et al. "Programming is hard-or at least it used to be: Educational opportunities and challenges of ai code generation". In: *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1.* 2023, pp. 500–506.

[3]   Tom Brown et al. "Language models are few-shot learners". In: *Advances in neural information processing systems* 33 (2020), pp. 1877–1901.

[4]   Mark Chen et al. "Evaluating large language models trained on code". In: *arXiv preprint arXiv:2107.03374* (2021).

[5]   Arghavan Moradi Dakhel et al. "Github copilot ai pair programmer: Asset or liability?" In: *Journal of Systems and Software* 203 (2023), p. 111734.

[6]   Patrick Diehl et al. "Benchmarking the parallel 1d heat equation solver in Chapel, Charm++, C++, HPX, Go, Julia, Python, Rust, Swift, and Java". In: *European Conference on Parallel Processing*. Springer. 2023, pp. 127–138.

[7]   *List of Top AI Code Generation Tools 2024*. https://www.trustradius.com/ai-code-generation. [Accessed 03-05-2024].

[8]   Yue Liu et al. "Refining ChatGPT-generated code: Characterizing and mitigating code quality issues". In: *ACM Transactions on Software Engineering and Methodology* (2023).

[9]   Julian Miller et al. "Applicability of the software cost model COCOMO II to HPC projects". In: *International Journal of Computational Science and Engineering* 17.3 (2018), pp. 283–296.

[10] Nhan Nguyen and Sarah Nadi. "An empirical evaluation of GitHub copilot's code suggestions". In: *Proceedings of the 19th International Conference on Mining Software Repositories*. 2022, pp. 1–5.

[11] *OpenAI Codex — openai.com*. https://openai.com/blog/openai-codex. [Accessed 14-05-2024].

[12] Hammond Pearce et al. "Can openai codex and other large language models help us fix security bugs?" In: *arXiv preprint arXiv:2112.02125* (2021).

[13] Julian Aron Prenner, Hlib Babii, and Romain Robbes. "Can OpenAI's codex fix bugs? an evaluation on QuixBugs". In: *Proceedings of the Third International Workshop on Automated Program Repair*. 2022, pp. 69–75.

[14] Jonathan Richard Shewchuk et al. "An introduction to the conjugate gradient method without the agonizing pain". In: (1994).

[15] Richard D Stutzke and May96 Crosstalk. *Software estimating technology: A survey*. Los. Alamitos, CA: IEEE Computer Society Press, 1997.

[16] Viriya Taecharungroj. ""What can ChatGPT do?" Analyzing early reactions to the innovative AI chatbot on Twitter". In: *Big Data and Cognitive Computing* 7.1 (2023), p. 35.

[17] Ningzhi Tang et al. "An empirical study of developer behaviors for validating and repairing ai-generated code". In: *13th Workshop on the Intersection of HCI and PL*. 2023.

[18] *TIOBE Index - TIOBE — tiobe.com*. https://www.tiobe.com/tiobe-index/. [Accessed 03-05-2024].

[19] Yuntao Wang et al. "A survey on ChatGPT: AI-generated contents, challenges, and solutions". In: *IEEE Open Journal of the Computer Society* (2023).

[20] Jiayang Wu et al. "Ai-generated content (aigc): A survey". In: *arXiv preprint arXiv:2304.06632* (2023).

[21] Beiqi Zhang et al. "Practices and challenges of using GitHub copilot: An empirical study". In: *arXiv preprint arXiv:2303.08733* (2023).