algorithms

MDPI

Article

Program Code Generation with Generative AIs

Baskhad Idrisov and Tim Schlippe *

IU International University of Applied Sciences, 99084 Erfurt, Germany; baskhad.idrisov@gmail.com
* Correspondence: tim.schlippe@iu.org

Abstract: Our paper compares the *correctness*, *efficiency*, and *maintainability* of human-generated and AI-generated program code. For that, we analyzed the computational resources of AI- and human-generated program code using metrics such as *time* and *space complexity* as well as *runtime* and *memory usage*. Additionally, we evaluated the *maintainability* using metrics such as *lines of code*, *cyclomatic complexity*, *Halstead complexity* and *maintainability index*. For our experiments, we had generative AIs produce program code in Java, Python, and C++ that solves problems defined on the competition coding website *leetcode.com*. We selected six LeetCode problems of varying difficulty, resulting in 18 program codes generated by each generative AI. GitHub Copilot, powered by Codex (GPT-3.0), performed best, solving 9 of the 18 problems (50.0%), whereas CodeWhisperer did not solve a single problem. BingAI Chat (GPT-4.0) generated *correct* program code for seven problems (38.9%), ChatGPT (GPT-3.5) and Code Llama (Llama 2) for four problems (22.2%) and StarCoder and InstructCodeT5+ for only one problem (5.6%). Surprisingly, although ChatGPT generated only four *correct* program codes, it was the only generative AI capable of providing a *correct* solution to a coding problem of difficulty level *hard*. In summary, 26 AI-generated codes (20.6%) solve the respective problem. For 11 AI-generated *incorrect* codes (8.7%), only minimal modifications to the program code are necessary to solve the problem, which results in time savings between 8.9% and even 71.3% in comparison to programming the program code from scratch.

Keywords: artificial intelligence; generative AIs; AI program code generation; program code efficiency; program code maintainability

check for updates

Citation: Idrisov, B.; Schlippe, T. Program Code Generation with Generative AIs. *Algorithms* **2024**, *17*, 62. https://doi.org/10.3390/a17020062

Academic Editors: Ulrich Kerzel, Mostafa Abbaszadeh, Andres Iglesias and Akemi Galvez Tomida

Received: 20 December 2023
Revised: 24 January 2024
Accepted: 26 January 2024
Published: 31 January 2024

1. Introduction

Generative AIs have increasingly become an integral part of our everyday lives [1]. Particularly, generative AIs for text generation are engineered to replicate human-like dialogues and offer help, information, and even emotional support [2–6]. They can be available 24/7 and provide instant responses, making them a valuable tool for customer service, personal assistants, and many other applications. Among them, ChatGPT (https://openai.com/chatgpt, accessed on 29 January 2024) stands out as one of the most frequently utilized for text generation [7]. Within just five days of its launch, more than one million users registered [7].

The transformative power of AI also permeates the field of coding, an area where technology has made significant strides: AI-powered chatbots are not only capable of conducting human-like conversations but can also generate program code [8,9]. Simultaneously, Integrated Development Environments (IDEs), unit tests, and benchmarking tools have simplified coding, making it more accessible to both seasoned developers and beginners [10]. In this sense, AI-powered chatbots and coding tools are two facets of the same coin, both contributing to the transformation of how we use technology. However, a critical challenge remains—the time and manual effort required for coding, especially for those new to the craft [11–13] .

While AI tools promise to streamline the coding process and lower the barriers to entry for aspiring coders, it is important to note that they still have their limits. For example, they may not always produce *correct*, *efficient*, or *maintainable* program codes. Therefore, it is crucial to carefully consider various aspects when deciding whether or not to use these tools [14].

While substantial research has focused on evaluation metrics for *correctness* like *pass@k* [15–17], there is a noticeable gap: The extensive comparison of AI-generated and human-generated program codes based on various metrics has largely been understudied. Consequently, our study embarks on a comprehensive and in-depth exploration of the coding capabilities of seven state-of-the-art generative AIs. Our goal is to evaluate the AI-generated program codes based on the interaction of various metrics such as *cyclomatic complexity*, *maintainability index*, etc., concerning their *correctness*, *efficiency* and *maintainability*. Moreover, we go one step further by comparing the AI-generated codes with corresponding human-generated codes written by professional programmers.

Our contributions are as follows:

- We investigate and compare the *correctness*, *efficiency* and *maintainability* of AI-generated program codes using varying evaluation metrics.
- We are the first to extensively compare AI-generated program codes to human-generated program codes.
- We analyze the program codes that address problems of three difficulty levels—*easy*, *medium* and *hard*.
- We produced a dataset of 126 AI-generated and 18 human-generated program codes—the *AI/Human-Generated Program Code Dataset*—which we share with the research community (https://github.com/Back3474/AI-Human-Generated-Program-Code-Dataset, accessed on 29 January 2024).

In the next section, we will provide an overview of related work. In Section 3, we will present our experimental setup. Our experiments and results will be described in Section 4. In Section 5, we will conclude our work and indicate possible future steps.

2. Related Work

In this section, we will look at existing work regarding automatic program code generation and evaluation.

Codex (https://openai.com/blog/openai-codex, accessed on 29 January 2024) is an AI model for program code generation based on GPT-3.0 and fine-tuned on program code publicly available on GitHub [16]. Ref. [16] tested Codex's capabilities in generating Python code using natural language instructions found in in-code comments known as docstrings. They also created *HumanEval*, a dataset of 164 hand-written coding problems in natural language plus their unit tests to assess the *functional correctness* of program code. One discovery from their research was that if Codex is asked to generate code for the same problem several times, the probability that one generated code is *correct* increases. Consequently, they used *pass@k* as an evaluation metric, where *k* is the number of generated program codes, and *pass* is the number of tasks, of which all unit tests were passed. To obtain an unbiased estimation of *pass@k*, ref. [16] applied additional adjustments to the original calculation. Codex achieved a *pass*@1 of 28.8% in solving the provided problems. They compared the Codex-generated code with program code generated by GPT-3.0 and GPT-J [18], but GPT-3.0 demonstrated a *pass*@1 of 0% and GPT-J obtained 11.4%. With *pass*@100, Codex achieved even 70.2%.

Ref. [19] evaluated *the validity*, *correctness*, and *efficiency* of program code generated by GitHub (GH) Copilot using *HumanEval*. GH Copilot is an IDE extension that uses Codex. Ref. [19] defined a code as *valid*, if it was compliant with the syntax rules of a given programming language. The *correctness* was computed by dividing the programming tasks' passed unit tests by all existing unit tests for this specific task. The *efficiency* was measured by determining the *time* and *space complexities* of the generated codes. Their results demonstrate that Codex was able to generate *valid* code with a success rate of 91.5%. Regarding code *correctness*, 28.7% were generated *correctly*, 51.2% were generated *partially correctly*, and 20.1% were generated *incorrectly*.

Ref. [17] used *HumanEval* to evaluate the *validity*, *correctness*, *security*, *reliability*, and *maintainability* of Python code generated by GH Copilot, Amazon CodeWhisperer (https://aws.amazon.com/de/codewhisperer, accessed on 29 January 2024), and ChatGPT. They defined a *valid* code and a *correct* code as in [19]. To determine the *security*, *reliability*, and *maintainability*, they used SonarQube (https://www.sonarqube.org, accessed on 29 January 2024). Their calculation of a code's *security* is based on the potential cybersecurity vulnerabilities of this code. The *reliability* is based on the number of bugs. The *maintainability* is measured by counting present code smells. ChatGPT generated 93.3% *valid* program codes, GH Copilot 91.5%, and CodeWhisperer 90.2%. ChatGPT passed most unit tests by generating 65.2% of *correct* program codes. GH Copilot reached 46.3%, and CodeWhisperer 31.1%. But when they evaluated newer versions of GH Copilot and CodeWhisperer, they measured 18% and 7% better values for *correctness*. Due to the small number of generated code fragments, the numbers for *security* were not usable. Concerning *reliability*, ChatGPT produced two bugs, GH Copilot three bugs and CodeWhisperer one bug. CodeWhisperer produced the most *maintainable* code, ChatGPT the second most, and GH Copilot the least *maintainable*.

Ref. [15] introduced a new framework for program code evaluation named *EvalPlus*. Furthermore, they created *HumanEval+*, an extension of *HumanEval* using *EvalPlus' automatic test input generator*. *HumanEval+* is 80 times larger than *HumanEval* which enables more comprehensive testing and analysis of AI-generated code. With *HumanEval+*, ref. [15] evaluated the *functional correctness* of program code generated by 26 different AI models which are based on GPT-4 [20], Phind-CodeLlama [21], WizardCoder-CodeLlama [22], ChatGPT [23], Code Llama [24], StarCoder [25], CodeGen [26], CODET5+ [27], MISTRAL [28], CodeGen2 [29], VICUNA [30], SantaCoder [31], INCODER [32], GPT-J [33], GPT-NEO [34], PolyCoder [35], and StableLM [36] with *pass@k*. Looking at *pass*@1, the top five performers were GPT-4 (76.2%), Phind-CodeLlama (67.1%), WizardCoder-CodeLlama (64.6%), ChatGPT (63.4%), and Code Llama (42.7%).

DeepMind developed an AI model for code generation named *AlphaCode* [37]. On the coding competition website *codeforces.com*, the AI model achieved an average ranking in the top 54.3% of more than 5000 participants for Python and C++ tasks. The ranking takes *runtime* and *memory usage* into account.

Ref. [38] evaluated GH Copilot's ability to produce solutions for 33 LeetCode problems using Python, Java, JavaScript, and C. They calculated the *correctness* by dividing the passed test cases by the total number of test cases per problem. The numbers for *understandability*, *cyclomatic* and *cognitive complexity* were retrieved using SonarQube, but due to configuration problems, they were not able to analyze the *understandability* of the codes generated in C. Concerning *correctness*, GH Copilot performed best in Java (57%) and worst in JavaScript (27%). In terms of *understandability*, GH Copilot's Python, Java and JavaScript program codes had a *cognitive complexity* of 6 and a *cyclomatic complexity* of 5 on average, with no statistically significant differences between the programming languages.

In comparison to the aforementioned related work, our focus is to evaluate the Java, Python and C++ code produced by Codex (GPT-3.0), CodeWhisperer, BingAI Chat (GPT-4.0), ChatGPT (GPT-3.5), Code Llama (Llama 2), StarCoder, and InstructCodeT5+. For comparison, we also assess human-generated program code. We obtain the *correctness*, *efficiency* and *maintainability* for our program codes by measuring *time* and *space complexity*, *runtime* and *memory usage*, *lines of code*, *cyclomatic complexity*, *Halstead complexity* and *maintainability index*. As a user usually does not generate code for the same problem several times, we evaluate *pass@k* with $k = 1$, which is our metric for *correctness*. Finally, we analyze which of the *incorrect* and the *not executable* program codes have the potential to be easily modified manually and then used quickly and without much effort to solve the corresponding coding problems.

3. Experimental Setup

In this section, we will provide a comprehensive overview of our experimental setup. This includes an introduction to the state-of-the-art generative AIs that we employed in our experiments, the coding problems we chose from LeetCode and the code quality metrics we utilized for evaluation.

3.1. Generative AI Models

This section introduces the generative AIs which we analyzed in our study.

3.1.1. ChatGPT Powered by GPT-3.5

The generative AI ChatGPT uses the large language model (LLM) GPT-3.5 which was pre-trained on text and program code. It was first fine-tuned with labeled text data collected from human annotators and a proprietary dataset of question-answer pairs covering various chat topics and scenarios [23]. Then, it was fine-tuned with a reward model (Reinforcement Learning from Human Feedback) and the Proximal Policy Optimization algorithm [23]. ChatGPT leverages GPT-3.5's 175 billion parameters [39] and is capable of generating program code in a wide variety of programming languages, including Python, JavaScript, HTML, CSS, SQL, Java, C#, C++, Ruby, PHP, R, Swift, and more [40].

3.1.2. BingAI Chat Powered by GPT-4.0

The generative AI BingAI Chat leverages the LLM GPT-4.0 which was trained on a text corpus of about 13 trillion tokens. Some of these tokens come from well-known datasets such as *CommonCrawl* and *RefinedWeb*, while others come from sources that are not publicly communicated [41,42]. GPT-4.0 was first fine-tuned with data sourced from ScaleAI plus text data from OpenAI. Then, it was fine-tuned with a reward model (Reinforcement Learning from Human Feedback) and the Proximal Policy Optimization algorithm [42,43]. It is estimated that the model has about 1.8 trillion parameters [41,42]. As it is not publicly known what programming languages are covered, we asked BingAI Chat "What programming languages are you capable of?". The answer was "I am capable of generating code in various programming languages, such as Python, Java, C#, JavaScript, Ruby, and more".

3.1.3. GitHub Copilot Powered by GPT-3.0's Fine-Tuned Model Codex

The generative AI GH Copilot uses GPT-3.0's fine-tuned LLM Codex. It is an IDE extension available in Visual Studio, Neovim, VS Code and JetBrains [44]. Codex was fine-tuned on natural language and a vast amount of lines of code from public sources, including GitHub repositories [45]. Codex was pre-trained on the datasets *Common Crawl (filtered)*, *WebText2*, *Books1*, *Books2* and *Wikipedia*. It has 175 billion parameters [46]. According to [45], Codex performs best in Python, but it also shows proficient results in other programming languages including JavaScript, Go, Perl, PHP, Ruby, Swift and TypeScript.

3.1.4. StarCoder Powered by StarCoderBase

The generative AI StarCoder leverages the LLM StarCoderBase. Like GH Copilot, StarCoder is available by using an extension in VS Code, Neovim, Jupyter [47] or all IntelliJ-based IDEs [48]. According to [25], StarCoderBase was first trained on permissively licensed data from GitHub, including 80+ programming languages, Git commits, GitHub issues, and Jupyter notebooks. Then, it was fine-tuned "on 35B Python tokens, resulting in the creation of StarCoder." which has 15.5 billion parameters [25]. StarCoder is capable of 86 programming languages, including Python, C++, Java, Kotlin, PHP, Ruby, TypeScript, and others.

3.1.5. Code Llama Powered by Llama 2

The generative AI Code Llama is based on Meta's LLM Llama 2 and was fine-tuned on 500B tokens. For our experiments, we took the version with 13 billion parameters (https://huggingface.co/codellama/CodeLlama-13b-hf, accessed on 29 January 2024) accessing via the same VS Code extension as for StarCoder [47]. The training data were mostly deduplicated, publicly available code, with 8% from natural language datasets related to program code. It covers many programming languages such as Python, C++, Java, PHP, TypeScript, JavaScript, C#, and Bash [24].

3.1.6. CodeWhisperer

According to [49], Amazon's "CodeWhisperer is a generative AI service powered by a foundation model trained on various data sources, including Amazon and open-source code". No information regarding CodeWhisperer's training, fine-tuning or number of parameters is publicly available. Supported IDEs to access CodeWhisperer are Amazon SageMaker Studio, JupyterLab, VS Code, all JetBrains IDEs, AWS Cloud9, AWS Lambda and AWS Glue Studio. Supported programming languages are Java, Python, JavaScript, TypeScript, C#, Ruby, Go, PHP, C++, C, Shell, Scala, Rust, Kotlin, and SQL [50].

3.1.7. InstructCodeT5+ 16b Powered by CodeT5+

The generative AI InstructCodeT5+ 16b leverages the LLM CodeT5+ and was pre-trained using the datasets *CodeSearchNet* [51] and *github-code* [52]. Then, it was fine-tuned on the dataset *CodeAlpaca* [53]. InstructCodeT5+ has 16 billion parameters and is capable of producing code in C, C++, C#, Go, Java, JavaScript, PHP, Python, and Ruby [27]. Since InstructCodeT5+ 16b was not available via chat or IDE extension, we prompted it via Google Colab and HuggingFace.

3.2. Our AI/Human-Generated Program Code Dataset

In this subsection, we will first introduce the coding platform LeetCode. Then we will present the coding problems that we selected from LeetCode for our experiments.

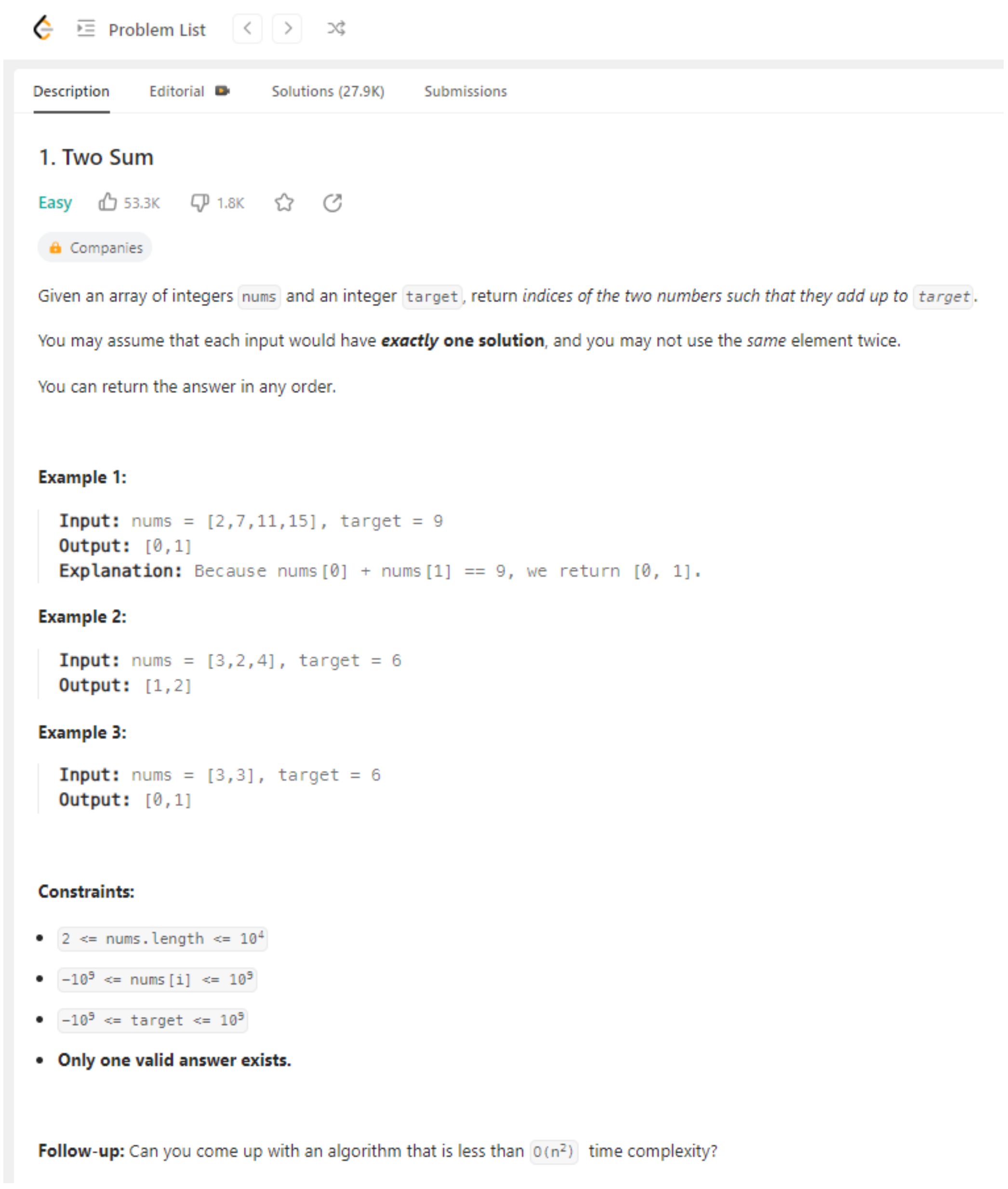
3.2.1. LeetCode as a Source for Coding Problems

LeetCode is an online platform where users can improve their coding skills [54]. The coding platform offers support for various programming languages, such as Java, Python, and C++ [38] which was optimal for our experiments. It has gained popularity among job seekers and coding enthusiasts as a resource for technical interviews and coding competitions. The platform offers a vast amount of technical resources and almost 3000 coding problems of six categories (*algorithms*, *database*, *Pandas*, *JavaScript*, *Shell*, *concurrency*) in three difficulty levels (*easy*, *medium*, *hard*).

As shown in Figure 1, each coding problem consists of (1) a textual description of the coding task, (2) examples with input values, expected outputs and sometimes explanations and (3) constraints, which for example define the range of variables' values. Once the user submits code to LeetCode which solves the problem, LeetCode reports corresponding *runtime* and *memory usage*. In case the submitted code is *not executable*, an error message appears.

3.2.2. Selected Coding Problems for Our Evaluation

For our experiments, we picked six LeetCode coding problems of the category *algorithms* that the generative AIs had to solve in Java, Python, and C++: Two coding problems were of the difficulty level *easy*, two *medium* and two *hard*. We chose the category *algorithms* to evaluate *efficiency* and *maintainability* since it is the category whose program code—if not programmed well—takes up a lot of memory and runtime and is very difficult to understand compared to the other LeetCode categories. To minimize the likelihood that the solutions to the coding problems appear in the training data of the generative AIs, we used the latest coding problems provided by LeetCode. The combination of coding problems and the number of evaluated generative AIs resulted in a total of 126 AI-generated codes (six problems · seven generative AIs · three languages = 126 codes). Table 1 provides an overview of the final selection of coding problems with their difficulty levels. For comparing the *efficiency* and *maintainability* of AI- and human-generated program codes, we chose the human-generated Java, Python and C++ program codes from LeetCode that solve our coding problems and were rated best by the LeetCode community, i.e., were written by programmers with the highest expertise level.



1. Two Sum

Given an array of integers `nums` and an integer `target`, return *indices of the two numbers such that they add up to* `target`.

You may assume that each input would have ***exactly* one solution**, and you may not use the *same* element twice.

You can return the answer in any order.

Example 1:

```
Input: nums = [2,7,11,15], target = 9
Output: [0,1]
Explanation: Because nums[0] + nums[1] == 9, we return [0, 1].
```

Example 2:

```
Input: nums = [3,2,4], target = 6
Output: [1,2]
```

Example 3:

```
Input: nums = [3,3], target = 6
Output: [0,1]
```

Constraints:

- `2 <= nums.length <=` 10^4
- -10^9 `<= nums[i] <=` 10^9
- -10^9 `<= target <=` 10^9
- **Only one valid answer exists.**

Follow-up: Can you come up with an algorithm that is less than $O(n^2)$ time complexity?

Figure 1. LeetCode Example: Two Sum.

Table 1. Selected Coding Problems for our Evaluation.

#	Coding Problem	Difficulty Level
1	Max Pair Sum in an Array	Easy
2	Faulty Keyboard	Easy
3	Minimum Absolute Difference Between Elements With Constraint	Medium
4	Double a Number Represented as a Linked List	Medium
5	Apply Operations to Maximize Score	Hard
6	Maximum Elegance of a K-Length Subsequence	Hard

3.3. Evaluation Metrics

In this subsection, we will describe the evaluation metrics *lines of code*, *cyclomatic complexity*, *runtime and memory usage*, *time and space complexity*, *Halstead complexity*, and *Maintainability Index* which we used to evaluate the *efficiency* and *maintainability* of the generated codes and to estimate the time to estimate the *time to correct* the *incorrect* program codes manually. Our goal was to present a comprehensive picture of code quality.

3.3.1. Lines of Code

One metric that we used to measure the quality of the program code independent of a certain computational environment was the number of *lines of code* (*LOC*). *LOC* provides an initial indication of the *complexity* based on the length of the program code. A higher *LOC* often suggests a more intricate program code structure, which can imply a greater textual and logical *complexity*. Consequently, in our analysis, more *LOC* indicates worse code quality. To measure and visualize *LOC*, we used a vs. Code extension called Codalyze (https://github.com/selcukusta/codalyze-rest-api, accessed on 29 January 2024).

3.3.2. Cyclomatic Complexity

Our second metric to measure the quality of the program code independent of a certain computational environment was the *cyclomatic complexity*. The *cyclomatic complexity*, also known as McCabe's Complexity introduced in [55], measures the number of linearly independent paths through a program code. This helps understand the logical *complexity* of the code: A higher *cyclomatic complexity* means more test cases are needed to achieve good program code coverage, indicating worse code quality. Similar to *LOC*, we assessed and visualized this metric using Codalyze.

3.3.3. Time Complexity and Space Complexity

Time complexity and *space complexity* are metrics that also assess program code independently of a computational environment. These metrics quantify the upper bounds of time and space needed by an algorithm as a function of the input [56]. The lower the order of the function, the better the *complexity*. For example, concerning *time complexity*, an algorithm with two nested loops results in a quadratic function (expressed as $O(n^2)$), while an algorithm with a single loop indicates a linear and less complex function (expressed as $O(n)$). According to [56], pp. 45–47 and [57], we determined the *time* and *space complexities* manually.

3.3.4. Halstead Complexity

Halstead complexity describes various program code quality metrics that can be calculated statically, i.e., independently of a computational environment [58]. Table 2 lists the *Halstead complexity* metrics that we leveraged for our evaluation. The calculation of the individual *Halstead complexity* values necessitates the number of distinct operators (n_1), the total number of operators (N_1), the number of distinct operands (n_2), and the total number of operands (N_2) in the program code. As shown in Table 2, the *Halstead complexity* values provide detailed information about the program code in different dimensions, for which only the numbers of operators and operands are required. To determine these numbers in Java, we used *Java-Code-Analyzer* (https://github.com/Taha248/Java-Code-Analyzer, accessed on 29 January 2024). To determine the numbers of operators and operands in Python, we used *HalsteadMetrics4Py* (https://github.com/ArturoG1z/HalsteadMetrics4Py, accessed on 29 January 2024). To determine these numbers in C++, we used *Halstead* (https://github.com/SparshBansal/Halstead, accessed on 29 January 2024). Based on the *LOC*, *cyclomatic complexity* and *Halstead volume*, we were able to determine the *maintainability index*. Then, with the help of *the maintainability index* and the *implementation time (T)*, we developed the formula to calculate the *estimated time to correct (TTC)* the *incorrect* program code and retrieve the *correct* program code, which we will describe in Section 4.9.

Table 2. Calculation of *Halstead Metrics*.

Halstead Metrics	**Calculation**
Length of the program (N)	$N = N_1 + N_2$
Vocabulary of the program (n)	$n = n_1 + n_2$
Volume of the program (V)	$V = (N_1 + N_2) \log_2(n_1 + n_2)$
Difficulty of the program (D)	$D = (n_1/2) \cdot (N_2/n_2)$
Programming effort (E)	$E = D \cdot V$
Implementation time (T)	$T = E/S$ where $S = 18$

3.3.5. Maintainability Index

The *maintainability index* (*MI*) is designed to quantify the *maintainability* of code independent of a computational environment. For calculating the *maintainability index*, we used the following formula which expresses the *maintainability index* as a number between 0 and 100 and is proposed by [59]:

$$MI = MAX(0, (171 - 5.2 \cdot ln(HV) - 0.23 \cdot (CC) - 16.2 \cdot ln(LOC)) \cdot 100/171)$$

where *HV* is *Halstead volume*, *CC* is *cyclomatic complexity* and *LOC* is *lines of code* [59]. A higher *maintainability index* indicates program code with a higher *maintainability*.

3.3.6. Runtime and Memory Usage

To report the program code's *efficiency* in a computational environment, we measured *runtime* and *memory usage*. We retrieved the program code's *runtime* in milliseconds and *memory usage* in megabytes with the help of LeetCode, which executes the program code with varying input values and reports the maximum *runtime* and *memory usage* for these input values. LeetCode assures that with a high computational load on the platform by many users there can only be slight fluctuations in runtime and memory usage.

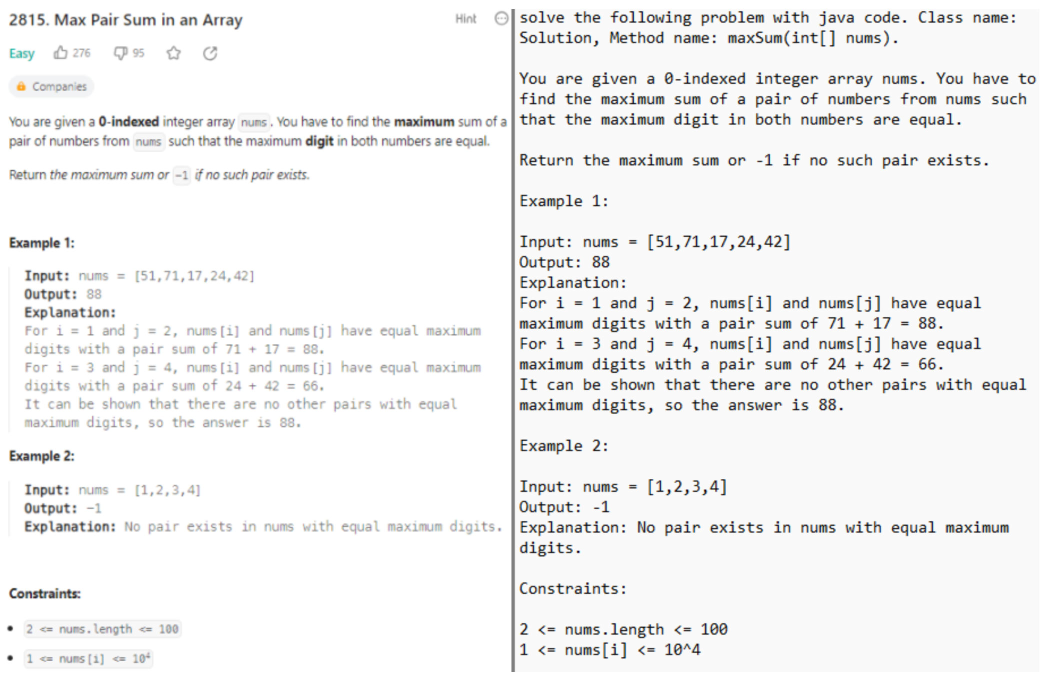
3.4. Prompt Engineering

For our tested generative AIs ChatGPT, Bing AI Chat, GH Copilot, StarCoder, Code Llama, CodeWhisperer, and InstructCodeT5+, we performed prompt engineering to find the prompts that led to optimal coding solutions in Java, Python and C++ for the six coding problems mentioned in Section 3.2.2. Figure 2 demonstrates our prompt which worked optimally for instructing ChatGPT to generate Java code (on the right) that solves the LeetCode coding problem (on the left).

Our prompt engineering strategy was to stick as close to LeetCode's coding problem description as possible and contained the following steps which are visualized in Figure 3:

1. We prompted "Solve the following problem with <programming language> code. <declarations>" together with the original LeetCode coding problem description, examples, example explanations and constraints.
2. If the result with prompt 1 was *incorrect* program code (i.e., did not solve the problem) or was *not executable*, we removed the example explanations.
3. If the result with prompt 2 was *incorrect* program code (i.e., did not solve the problem) or was *not executable*, we removed the examples.
4. If the result with prompt 3 was *incorrect* program code (i.e., did not solve the problem) or was *not executable*, we took the prompt from prompts 1–3 that resulted in program code that was closest to our human-generated reference.

In the next section, we will describe the quality of the program codes which were generated using the optimal prompt from this strategy.



2815. Max Pair Sum in an Array

Easy 276 95

Companies

You are given a **0-indexed** integer array `nums`. You have to find the **maximum** sum of a pair of numbers from `nums` such that the maximum **digit** in both numbers are equal.

Return the maximum sum or `-1` if no such pair exists.

Example 1:

```
Input: nums = [51,71,17,24,42]
Output: 88
Explanation:
For i = 1 and j = 2, nums[i] and nums[j] have equal maximum
digits with a pair sum of 71 + 17 = 88.
For i = 3 and j = 4, nums[i] and nums[j] have equal maximum
digits with a pair sum of 24 + 42 = 66.
It can be shown that there are no other pairs with equal
maximum digits, so the answer is 88.
```

Example 2:

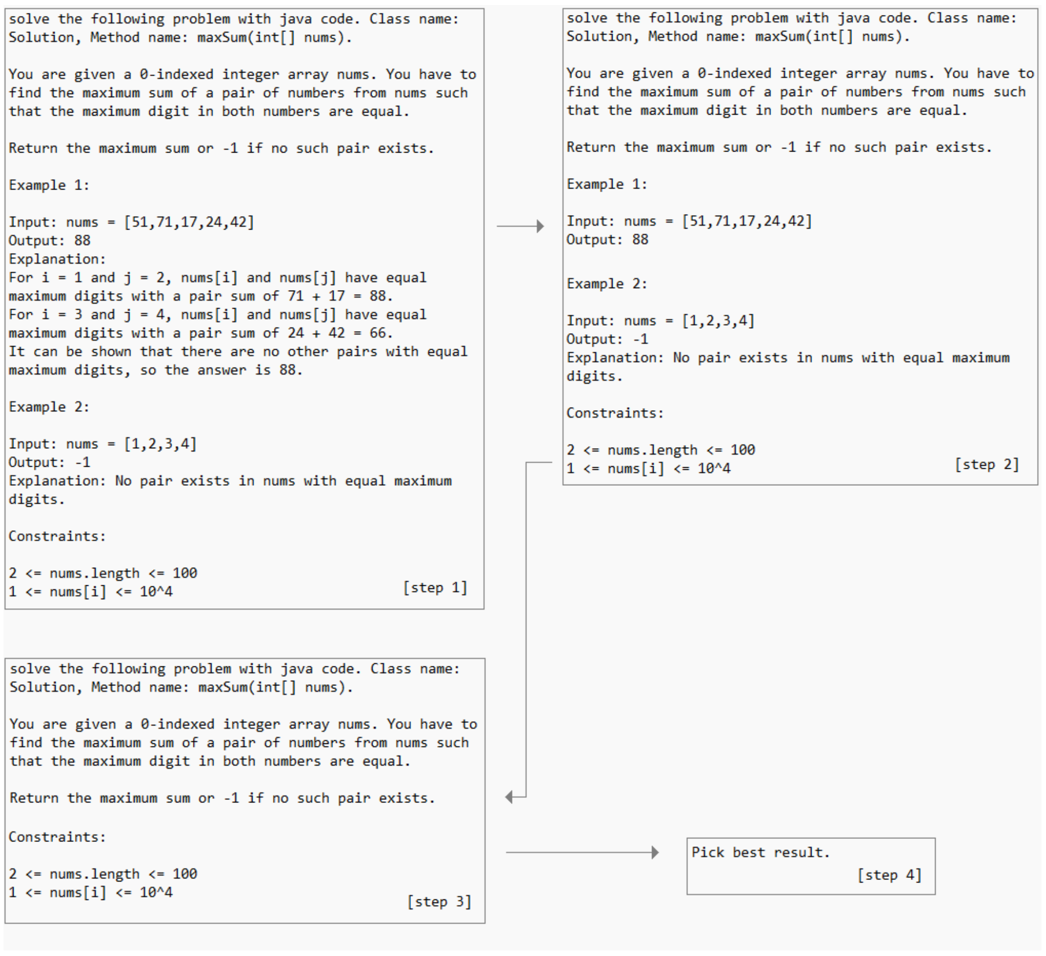
```
Input: nums = [1,2,3,4]
Output: -1
Explanation: No pair exists in nums with equal maximum digits.
```

Constraints:

- `2 <= nums.length <= 100`
- `1 <= nums[i] <= 10^4`

```
solve the following problem with java code. Class name:
Solution, Method name: maxSum(int[] nums).

You are given a 0-indexed integer array nums. You have to
find the maximum sum of a pair of numbers from nums such
that the maximum digit in both numbers are equal.

Return the maximum sum or -1 if no such pair exists.

Example 1:

Input: nums = [51,71,17,24,42]
Output: 88
Explanation:
For i = 1 and j = 2, nums[i] and nums[j] have equal
maximum digits with a pair sum of 71 + 17 = 88.
For i = 3 and j = 4, nums[i] and nums[j] have equal
maximum digits with a pair sum of 24 + 42 = 66.
It can be shown that there are no other pairs with equal
maximum digits, so the answer is 88.

Example 2:

Input: nums = [1,2,3,4]
Output: -1
Explanation: No pair exists in nums with equal maximum
digits.

Constraints:

2 <= nums.length <= 100
1 <= nums[i] <= 10^4
```

Figure 2. LeetCode Coding Problem Description (**left**) and Prompt to Instruct ChatGPT (**right**).



```
solve the following problem with java code. Class name:
Solution, Method name: maxSum(int[] nums).

You are given a 0-indexed integer array nums. You have to
find the maximum sum of a pair of numbers from nums such
that the maximum digit in both numbers are equal.

Return the maximum sum or -1 if no such pair exists.

Example 1:

Input: nums = [51,71,17,24,42]
Output: 88
Explanation:
For i = 1 and j = 2, nums[i] and nums[j] have equal
maximum digits with a pair sum of 71 + 17 = 88.
For i = 3 and j = 4, nums[i] and nums[j] have equal
maximum digits with a pair sum of 24 + 42 = 66.
It can be shown that there are no other pairs with equal
maximum digits, so the answer is 88.

Example 2:

Input: nums = [1,2,3,4]
Output: -1
Explanation: No pair exists in nums with equal maximum
digits.

Constraints:

2 <= nums.length <= 100
1 <= nums[i] <= 10^4                          [step 1]
```

```
solve the following problem with java code. Class name:
Solution, Method name: maxSum(int[] nums).

You are given a 0-indexed integer array nums. You have to
find the maximum sum of a pair of numbers from nums such
that the maximum digit in both numbers are equal.

Return the maximum sum or -1 if no such pair exists.

Example 1:

Input: nums = [51,71,17,24,42]
Output: 88

Example 2:

Input: nums = [1,2,3,4]
Output: -1
Explanation: No pair exists in nums with equal maximum
digits.

Constraints:

2 <= nums.length <= 100
1 <= nums[i] <= 10^4                          [step 2]
```

```
solve the following problem with java code. Class name:
Solution, Method name: maxSum(int[] nums).

You are given a 0-indexed integer array nums. You have to
find the maximum sum of a pair of numbers from nums such
that the maximum digit in both numbers are equal.

Return the maximum sum or -1 if no such pair exists.

Constraints:

2 <= nums.length <= 100
1 <= nums[i] <= 10^4                          [step 3]
```

```
Pick best result.
                  [step 4]
```

Figure 3. Prompt Engineering Strategy.

4. Experiments and Results

To ensure the validity of our experiments, we had all program codes generated by the generative AIs in the short time period from August to September 2023, making sure that for each generative AI only the same version is used. In this section, we will first analyze which generated program codes are *correct*, i.e., solve the coding problems. Then we will compare the quality of the *correct* program codes using our evaluation criteria *lines of code*, *cyclomatic complexity*, *time complexity*, *space complexity*, *runtime*, *memory usage*, and *maintainability index*. Finally, we will evaluate which of the *incorrect* program codes—due to their *maintainability* and proximity to the *correct* program code—have the potential to be easily modified manually and then used quickly and without much effort to solve the corresponding problems.

4.1. Correct Solutions

Table 3 demonstrates which generated program codes for our six tasks in the three programming languages Java (*J*), Python (*P*) and C++ (*C*) were *correct* (indicated with ✓), i.e., did solve the coding problem. The entry "—" indicates program code that was *incorrect*, i.e., did not solve the coding problem.

Table 3. Correct Solutions (Java \| Python \| C++): Generative AI vs. *human*.

	Easy		**Medium**		**Hard**		**Total**
	#1	**#2**	**#3**	**#4**	**#5**	**#6**	
	J | P | C	**J | P | C**	**J | P | C**	**J | P | C**	**J | P | C**	**J | P | C**	**J | P | C | ∑**
ChatGPT (GPT-3.5)	— | — | ✓	— | ✓ | ✓	— | — | —	— | — | —	— | — | —	— | ✓ | —	0 | 2 | 2 | 4
Bing AI (GPT-4.0)	— | ✓ | ✓	✓ | ✓ | ✓	— | ✓ | ✓	— | ϵ | —	— | ϵ | —	— | — | —	1 | **3** | 3 | 7
GH Copilot (GPT-3.0)	✓ | — | ✓	✓ | ✓ | ✓	✓ | — | ✓	✓ | — | ✓	— | — | —	— | — | —	**4** | 1 | **4** | **9**
StarCoder	— | — | —	✓ | — | —	— | — | —	— | — | —	— | — | —	— | — | —	1 | 0 | 0 | 1
CodeWhisperer	— | — | —	— | — | —	— | — | —	— | — | —	ϵ | — | —	— | — | —	0 | 0 | 0 | 0
Code Llama (Llama 2)	— | — | —	✓ | ✓ | ✓	✓ | — | —	— | — | —	— | ϵ | —	— | — | —	2 | 1 | 1 | 4
InstructCodeT5+	— | — | —	— | ✓ | —	— | — | —	— | — | —	— | — | —	— | — | —	0 | 1 | 0 | 1
human	✓ | ✓ | ✓	✓ | ✓ | ✓	✓ | ✓ | ✓	✓ | ✓ | ✓	✓ | ✓ | ✓	✓ | ✓ | ✓	6 | 6 | 6 | 18

We observe that 122 of the 126 AI-generated program codes are *executable*. Four program codes result in a compilation error which is indicated with ϵ. Out of our 18 programming tasks, *GH Copilot* was able to generate nine *correct* program codes (50%) in total (∑), followed by *Bing AI* with seven *correct* program codes (39%). *ChatGPT* and *Code Llama* produced only four *correct* program codes (22%). *CodeWhisperer* did not produce *correct* program code in all three programming languages. This results in 26 *correct* program codes, which we will further analyze in Sections 4.2–4.8.

Looking at the difficulty levels shows that the generative AIs rather generated *correct* program code for *easy* and *medium* coding problems. Regarding the programming languages, most *correct* Java (*J*) and C++ (*C*) code were generated with *GH Copilot*. Most *correct* Python (*P*) code was produced with *Bing AI*. The program code which was written by human programmers (*human*) was always *correct*, independent of the programming language and the difficulty of the coding problem.

4.2. Lines of Code

Table 4 shows the number of code lines in the *correct* 26 program codes which solve the corresponding coding problem plus the number of code lines in our human-written reference program codes (*human*).

Out of the 26 *correct* program codes, in eight cases (31%) a generative AI was able to solve the coding problem with code that contains fewer *lines of code* than *human*. However, 13 AI-generated program codes (50%) were outperformed by *human* in this evaluation metric. For task#2, all five corresponding program codes (19%) contained the same number of *lines of code* as *human*. *BingAI* and *GH Copilot* performed better than the other generative AIs, being able to generate three program codes with fewer *lines of code* than *human*.

Table 4. Lines of Code (Java | Python | C++): Generative AI vs. *human*.

	Easy						**Medium**						**Hard**					
	#1			**#2**			**#3**			**#4**			**#5**			**#6**		
	J	**P**	**C**	**J**	**P**	**C**	**J**	**P**	**C**	**J**	**P**	**C**	**J**	**P**	**C**	**J**	**P**	**C**
ChatGPT (GPT-3.5)	—	—	18	—	**8**	11	—	—	—	—	—	—	—	—	—	—	**10**	—
Bing AI (GPT-4.0)	—	13	19	**11**	**8**	11	—	**7**	**9**	—	—	—	—	—	—	—	—	—
GH Copilot (GPT-3.0)	19	—	20	**11**	**8**	14	**10**	—	**10**	45	—	21	—	—	—	—	—	—
StarCoder	—	—	—	12	—	—	—	—	—	—	—	—	—	—	—	—	—	—
CodeWhisperer	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
Code Llama (Llama 2)	—	—	—	12	**8**	12	**10**	—	—	—	—	—	—	—	—	—	—	—
InstructCodeT5+	—	—	—	—	**8**	—	—	—	—	—	—	—	—	—	—	—	—	—
human	**18**	**9**	**12**	15	**8**	**9**	11	13	11	**10**	**10**	**10**	**69**	**43**	**60**	**20**	16	**23**
best AI vs. *human* (Δ in %)	−5	−31	−33	**+36**	0	−18	**+10**	**+86**	**+22**	−78	—	−52	—	—	—	—	**+60**	—

In the *hard* task#6, where only *ChatGPT* produced a *correct* program code, *ChatGPT* was even able to solve the problem with 10 Python (*P*) *lines of code*, which is 60% fewer *lines of code* than *human*. *BingAI* generated 86% fewer *lines of code* to solve task#3 in Python (*P*). *BingAI* and *GH Copilot* produced 36% fewer *lines of code* to solve task#2 in Java (*J*); 22% fewer C++ (*C*) *lines of code* are required in *BingAI*'s program code for task#3. Furthermore, 10% fewer Java (*J*) *lines of code* are required in *GH Copilot*'s and *Code Llama*'s program code for task#3.

4.3. Cyclomatic Complexity

Table 5 shows the *cyclomatic complexity* in the 26 *correct* program codes which solve the coding problem plus the *cyclomatic complexity* in our human-written reference program codes (*human*). The *cyclomatic complexity* measures the number of linearly independent paths through a program code. As described in Section 3.3.2, a higher *cyclomatic complexity* indicates worse code quality.

Table 5. Cyclomatic Complexity (Java | Python | C++): Generative AI vs. *human*.

	Easy						**Medium**						**Hard**					
	#1			**#2**			**#3**			**#4**			**#5**			**#6**		
	J	**P**	**C**	**J**	**P**	**C**	**J**	**P**	**C**	**J**	**P**	**C**	**J**	**P**	**C**	**J**	**P**	**C**
ChatGPT (GPT-3.5)	—	—	**4**	—	**3**	**3**	—	—	—	—	—	—	—	—	—	—	**4**	—
Bing AI (GPT-4.0)	—	6	6	**3**	**3**	**3**	—	**3**	**3**	—	—	—	—	—	—	—	—	—
GH Copilot (GPT-3.0)	7	—	**4**	**3**	**3**	**3**	**3**	—	**3**	9	—	**4**	—	—	—	—	—	—
StarCoder	—	—	—	**3**	—	—	—	—	—	—	—	—	—	—	—	—	—	—
CodeWhisperer	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
Code Llama (Llama 2)	—	—	—	**3**	**3**	**3**	**3**	—	—	—	—	—	—	—	—	—	—	—
InstructCodeT5+	—	—	—	—	**3**	—	—	—	—	—	—	—	—	—	—	—	—	—
human	**6**	**4**	**4**	**3**	**3**	**3**	4	5	4	**5**	**5**	5	**18**	**18**	**18**	**6**	6	**6**
best AI vs. *human* (Δ in %)	−14	−33	0	0	0	0	**+33**	**+67**	**+33**	−44	—	**+25**	—	—	—	—	**+50**	—

Out of the 26 *correct* program codes, in seven cases (27%) a generative AI was able to solve the coding problem with program code that has less *cyclomatic complexity* than *human*. Only four AI-generated program codes (15%) were outperformed by *human* in this evaluation metric. Fifteen program codes (58%) contain the same *cyclomatic complexity* as *human*. *GH Copilot* performed better than the other generative AIs being able to generate three program codes with less *cyclomatic complexity* than *human*.

In the *medium* task#3, where only *Bing AI* produced *correct* program code in Python (*P*), *Bing AI* was even able to solve the problem with a *cyclomatic complexity* of 3, which is 67% less than *human*. *ChatGPT* generated code with 50% less *cyclomatic complexity* to solve task#6 in Python (*P*). *GH Copilot* and *Code Llama* produced code with 33% less *cyclomatic complexity* to solve task#3 in Java (*J*). *Bing AI* and *GH Copilot* also generated code with 33% lower *cyclomatic complexity* to solve task#3 in C++ (*C*). Moreover, 25% less *cyclomatic complexity* is in the C++ code (*C*) for task#4.

4.4. Time Complexity

Table 6 illustrates the *time complexity* in the 26 *correct* program codes plus the *time complexity* in our human-written reference program codes (*human*). The *time complexity* quantifies the upper bound of time needed by an algorithm as a function of the input [56] as described in Section 3.3.3. The lower the order of the function, the better the *complexity*. The cross-column entries mean that the value is the same in all cross-columns.

Table 6. Time Complexity (Java | Python | C++): Generative AI vs. *human*.

	Easy						**Medium**						**Hard**					
	#1			**#2**			**#3**			**#4**			**#5**			**#6**		
	J	**P**	**C**	**J**	**P**	**C**	**J**	**P**	**C**	**J**	**P**	**C**	**J**	**P**	**C**	**J**	**P**	**C**
ChatGPT (GPT-3.5)	—	—	**n**	—	$\mathbf{n^2}$	n^2	—	—	—	—	—	—	—	—	—	—	$n^k k$	—
Bing AI (GPT-4.0)	—	nlogn	$n^2 k$	$\mathbf{n^2}$	$\mathbf{n^2}$	n^2	—	n^2	n^2	—	—	—	—	—	—	—	—	—
GH Copilot (GPT-3.0)	$\mathbf{n^2 m}$	—	nk	$\mathbf{n^2}$	$\mathbf{n^2}$	n^2	n^2	—	n^2	**n**	—	**n**	—	—	—	—	—	—
StarCoder	—	—	—	$\mathbf{n^2}$	—	—	—	—	—	—	—	—	—	—	—	—	—	—
CodeWhisperer	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
Code Llama (Llama 2)	—	—	—	$\mathbf{n^2}$	$\mathbf{n^2}$	n^2	n^2	—	—	—	—	—	—	—	—	—	—	—
InstructCodeT5+	—	—	—	—	$\mathbf{n^2}$	—	—	—	—	—	—	—	—	—	—	—	—	—
human	nm +nlogn +d	**n**	**n**	$\mathbf{n^2}$	$\mathbf{n^2}$	**n**	**logn**	**logn**	**nlogn**	**n**	**n**	**n**	**nlogn +mlog (logm)**			**nlogn**		
best AI vs. *human* (*h*)	**AI**	h	—	—	—	h	h	h	h	—	h	—	h	h	h	h	h	h

Out of the 26 *correct* program codes, in one case (4%) (*AI*) a generative AI was able to solve the coding problem with code that has less *time complexity* than *human*. Thirteen AI-generated program codes (50%) were outperformed by *human* in this evaluation metric. Twelve program codes (46%) contain the same *time complexity* as *human*. *GH Copilot* performed better than the other generative AIs being able to generate one program code with lower *time complexity* than *human* and four program codes with equal *time complexity*. *ChatGPT*, *Bing AI*, and *Code Llama* produced program code with the same *time complexity* as *human* in two program codes each.

4.5. Space Complexity

Table 7 illustrates the *space complexity* in the 26 *correct* program codes plus the *space complexity* in our human-written reference program codes (*human*). Similar to the *time complexity*, the *space complexity* quantifies the upper bound of space needed by an algorithm as a function of the input [57] as described in Section 3.3.3. Analogous to the *time complexity*, the lower the order of the function, the better the *complexity*. The cross-column entries also mean that the value is the same in all cross-columns.

Out of the 26 *correct* program codes, in seven cases (27%) (*AI*) a generative AI was able to solve the coding problem with code that has less *space complexity* than *human*. One AI-generated program code (4%) was outperformed by *human* in this evaluation metric. Eighteen program codes (69%) contain the same *space complexity* as *human*. This shows that the generative AIs perform significantly better in terms of *space complexity* compared to *time complexity*. Again, *GH Copilot* outperformed the other generative AIs being able to generate three program codes with lower *space complexity* than *human* and five program codes with equal *space complexity*. *Bing AI* generated program code with the same *space complexity* as *human* in five program codes, *ChatGPT* and *Code LLama* in three program codes each, as well as *StarCoder* and *InstructCodeT5+* in one program code each.

4.6. Runtime

Table 8 demonstrates the *runtime* of the 26 *correct* program codes plus the *runtime* of our human-written reference program codes (*human*) on LeetCode in milliseconds. The lower the *runtime* of a program code, the better. The *runtime* of the six *correct* program codes labeled with "*" could not be measured since their execution resulted in a *Time Limit Exceeded* error when executed on LeetCode.

Table 7. Space Complexity (Java | Python | C++): Generative AI vs. *human*.

	Easy						**Medium**						**Hard**					
	#1			**#2**			**#3**			**#4**			**#5**			**#6**		
	J	**P**	**C**	**J**	**P**	**C**	**J**	**P**	**C**	**J**	**P**	**C**	**J**	**P**	**C**	**J**	**P**	**C**
ChatGPT (GPT-3.5)	—	—	**1**	—	**n**	**n**	—	—	—	—	—	—	—	—	—	—	**k**	—
Bing AI (GPT-4.0)	—	**n**	**1**	**n**	**n**	**n**	—	**1**	**1**	—	—	—	—	—	—	—	—	—
GH Copilot (GPT-3.0)	**1**	—	k	**n**	**n**	**n**	**1**	—	**1**	**1**	—	**1**	—	—	—	—	—	—
StarCoder	—	—	—	**n**	—	—	—	—	—	—	—	—	—	—	—	—	—	—
CodeWhisperer	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
Code Llama (Llama 2)	—	—	—	**n**	**n**	**n**	**1**	—	—	—	—	—	—	—	—	—	—	—
InstructCodeT5+	—	—	—	—	**n**	—	—	—	—	—	—	—	—	—	—	—	—	—
human	n	**n**	**1**	**n**	**n**	**n**	n	n	n	**1**	**1**	**1**	**n + m**			**n**	n	**n**
best AI vs. *human* (*h*)	**AI**	—	—	—	—	—	**AI**	**AI**	**AI**	—	—	—	h	h	h	h	**AI**	h

Table 8. Runtime (Java | Python | C++): Generative AI vs. *human*.

	Easy						**Medium**						**Hard**					
	#1			**#2**			**#3**			**#4**			**#5**			**#6**		
	J	**P**	**C**	**J**	**P**	**C**	**J**	**P**	**C**	**J**	**P**	**C**	**J**	**P**	**C**	**J**	**P**	**C**
ChatGPT (GPT-3.5)	—	—	16	—	37	4	—	—	—	—	—	—	—	—	—	—	*	—
Bing AI (GPT-4.0)	—	126	42	**3**	49	8	—	*	*	—	—	—	—	—	—	—	—	—
GH Copilot (GPT-3.0)	**4**	—	23	**3**	47	3	*	—	*	5	—	**209**	—	—	—	—	—	—
StarCoder	—	—	—	10	—	—	—	—	—	—	—	—	—	—	—	—	—	—
CodeWhisperer	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
Code Llama (Llama 2)	—	—	—	**3**	55	13	*	—	—	—	—	—	—	—	—	—	—	—
InstructCodeT5+	—	—	—	—	**33**	—	—	—	—	—	—	—	—	—	—	—	—	—
human	9	**107**	**11**	**3**	51	**0**	**107**	**1 k**	**213**	**2**	**365**	234	**390**	**7 k**	**718**	**46**	**1 k**	**458**
best AI vs. *human* (Δ in %)	**+125**	−15	−31	0	**+55**	−100	—	—	—	−60	—	**+12**	—	—	—	—	−100	—

* correct, but *Time Limit Exceeded* on LeetCode.

Out of the 26 *correct* program codes, in six cases (23%) a generative AI was able to solve the coding problem with code that executes with less *runtime* than *human*. Seventeen AI-generated program codes (65%) were outperformed by *human* in this evaluation metric. Three program codes (12%) took the same *runtime* as *human*. *GH Copilot* performed better than the other generative AIs being able to generate two program codes with less *runtime* than *human* and one program code with the same *runtime*.

In easy task#1, where only *GH Copilot* produced the *correct* program code in Java (*J*), *GH Copilot* was even able to solve the problem in a *runtime* of 4 milliseconds, which is 125% less than *human*. *InstructCodeT5+* generated code that took 55% less *runtime* to solve task#2 in Python (*P*). *GH Copilot* produced code that took 12% less *runtime* to solve task#4 in C++ (*C*). *Bing AI*, *GH Copilot* and *Code Llama* generated code that took the same *runtime* as *human* to solve task#2 in Java (*J*).

4.7. Memory Usage

Table 9 lists the *memory usage* of the 26 *correct* program codes plus the *memory usage* of our human-written reference program codes (*human*) on LeetCode in megabytes. The lower the *memory usage* of a program code, the better. The *memory usage* of the six *correct* program codes labeled with "*" could not be measured since their execution resulted in a *Time Limit Exceeded* error when executed on LeetCode.

Out of the 26 *correct* program codes, in five cases (19%) a generative AI was able to solve the coding problem with code that executes with less *memory usage* than *human*. Eleven AI-generated program codes (42%) were outperformed by *human* in this evaluation metric. Ten program codes (38%) took the same *memory usage* as *human*. *GH Copilot* performed better than the other generative AIs being able to generate one program code with less *memory usage* than *human* and five program codes with the same *memory usage*, closely followed by *Bing AI* which was able to generate one program code with less *memory usage* than *human* and three program codes with the same *memory usage*. However, in the AI-generated program codes, which have lower *memory usage*, the *memory usage* of less than 1% relative is not significantly lower than in *human*.

Table 9. Memory Usage (Java | Python | C++): Generative AI vs. *human*.

	Easy						**Medium**						**Hard**					
	#1			**#2**			**#3**			**#4**			**#5**			**#6**		
	J	**P**	**C**	**J**	**P**	**C**	**J**	**P**	**C**	**J**	**P**	**C**	**J**	**P**	**C**	**J**	**P**	**C**
ChatGPT (GPT-3.5)	—	—	**68**	—	**16**	7	—	—	—	—	—	—	—	—	—	—	∗	—
Bing AI (GPT-4.0)	—	**16**	**68**	**44**	**16**	7	—	∗	∗	—	—	—	—	—	—	—	—	—
GH Copilot (GPT-3.0)	**44**	—	69	**44**	**16**	**6**	∗	—	∗	**45**	—	**116**	—	—	—	—	—	—
StarCoder	—	—	—	45	—	—	—	—	—	—	—	—	—	—	—	—	—	—
CodeWhisperer	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
Code Llama (Llama 2)	—	—	—	**44**	**16**	7	∗	—	—	—	—	—	—	—	—	—	—	—
InstructCodeT5+	—	—	—	—	**16**	—	—	—	—	—	—	—	—	—	—	—	—	—
human	**44**	**16**	**68**	**44**	17	**6**	**57**	**32**	**118**	**45**	**20**	**116**	**57**	**39**	**230**	**89**	**53**	**180**
best AI vs. *human* (Δ in %)	0	0	0	0	**+1**	0	—	—	—	0	—	0	—	—	—	—	—	—

* correct, but *Time Limit Exceeded* on LeetCode.

4.8. Maintainability Index

Table 10 demonstrates the *maintainability index* of the 26 *correct* program codes plus the *maintainability index* of our human-written reference program codes (*human*). The higher the *maintainability index* of a program code, the better.

Table 10. Maintainability Index (Java | Python | C++): Generative AI vs. *human*.

	Easy						**Medium**						**Hard**					
	#1			**#2**			**#3**			**#4**			**#5**			**#6**		
	J	**P**	**C**	**J**	**P**	**C**	**J**	**P**	**C**	**J**	**P**	**C**	**J**	**P**	**C**	**J**	**P**	**C**
ChatGPT (GPT-3.5)	—	—	53	—	**64**	60	—	—	—	—	—	—	—	—	—	—	**59**	—
Bing AI (GPT-4.0)	—	56	52	**59**	**64**	60	—	**64**	**60**	—	—	—	—	—	—	—	—	—
GH Copilot (GPT-3.0)	**52**	—	51	**59**	63	57	**59**	—	**60**	41	—	51	—	—	—	—	—	—
StarCoder	—	—	—	58	—	—	—	—	—	—	—	—	—	—	—	—	—	—
CodeWhisperer	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
Code Llama (Llama 2)	—	—	—	58	**64**	59	**59**	—	—	—	—	—	—	—	—	—	—	—
InstructCodeT5+	—	—	—	—	**64**	—	—	—	—	—	—	—	—	—	—	—	—	—
human	51	**62**	**57**	56	63	**62**	56	56	57	**59**	**60**	**59**	**33**	**39**	**34**	**50**	53	**48**
best AI vs. *human* (Δ in %)	**+2**	−10	−7	**+5**	**+2**	−3	**+5**	**+14**	**+5**	−30	—	−14	—	—	—	—	**+11**	—

Out of the 26 *correct* program codes, in 13 cases (50%) a generative AI was able to produce code with a higher *maintainability index* than *human*. Thirteen AI-generated program codes (50%) were exceeded by *human* in this evaluation metric. *GH Copilot* and *Bing AI* performed better than the other generative AIs being able to generate four program codes (15%) with a higher *maintainability index* than *human*. *ChatGPT* and *Code Llama* produced two program codes (8%) with a higher *maintainability index* than *human*.

4.9. Potential of Incorrect AI-Generated Program Code

After analyzing the 26 *correct* program codes, our goal was to evaluate which of the 96 *incorrect* and the four *not executable* program codes have the potential to be easily modified manually and then used quickly and without much effort to solve the corresponding coding problems. Consequently, we first had programmers determine which of the 96 *incorrect* program codes have the potential to be corrected manually due to their *maintainability* and proximity to the *correct* program code. This resulted in 24 *potentially correct* program codes, for which we further estimated the *time to correct* (*TTC*) the *incorrect* program code and retrieve the *correct* program code.

In order to have a fair comparison between the program codes that are not dependent on the experience of any programmers, we report the *TTC* using Halstead's estimates of the *implementation time*, which is only dependent on the operators and operands in the program code—not on the expertise of a programmer. Consequently, to estimate the *TTC* in seconds, we developed the following formula:

$$TTC = |T_{correct} - T_{incorrect}| + T_{maintain}$$

where $T_{correct}$ is Halstead's *implementation time* ([58], pp. 57–59) of the *correct* program code in seconds and $T_{incorrect}$ is Halstead's *implementation time* ([58], pp. 57–59) of the *incorrect* program code in seconds. The use of the absolute difference is necessary because $T_{correct}$ can have a lower value than $T_{incorrect}$ if parts of the program code need to be removed in order to obtain the *correct* program code. As Healstead's *implementation time* only considers the time for the effort of implementing and understanding the program based on the operators and operands but not time to maintain the code—which is necessary when correcting program code—we additionally computed the time to maintain the program with the help of the *maintainability index MI*. The *MI* is based on *lines of code*, *cyclomatic complexity* and Halstead's *volume* as described in Section 3.3.5 and shown in Table 2. $T_{maintain}$ in seconds is estimated with the following formula:

$$T_{maintain} = \frac{T_{incorrect}}{MI/100} - T_{incorrect}$$

where MI is the *maintainability index* between 0 and 100, based on [60]. To obtain MI in a range of 0 to 1, we divided it by 100. This way, $T_{incorrect}$ is extended with a factor that is higher for less *maintainable* program code.

Table 11 shows the MI, $T_{incorrect}$, $T_{correct}$, TTC as well as the relative difference between $T_{correct}$ and TTC (Δ $T_{correct}$–TTC (%)) of our 24 *potentially correct* program codes and their corresponding *correct* program codes. We observe that for 11 program codes $TTC < T_{correct}$, i.e., the *time to correct* (TTC) the *incorrect* program code takes less time than the *implementation time* of the *correct* program code $T_{correct}$. With these 11 codes, between 8.85% and even 71.31% of time can be saved if the AI-generated program code is corrected and not programmed from scratch.

Table 11. Potential of *Incorrect* AI-Generated Program Code.

	#	Lang.	MI	Estimated Time To Program in Seconds		TTC	Δ $T_{correct}$–TTC (%)
				$T_{incorrect}$	$T_{correct}$		
StarCoder	1	C	49.77	1001	1225	1234	−0.73
CodeWhisperer	1	J	57.55	713	1167	1693	−31.07
StarCoder	1	J	57.78	827	1464	**1241**	**+17.97**
ChatGPT (GPT-3.5)	3	J	49.73	1570	2984	3001	−0.57
StarCoder	3	P	57.41	1045	450	1370	−67.15
CodeWhisperer	2	C	58.46	355	361	**258**	**+39.92**
ChatGPT (GPT-3.5)	3	P	60.48	303	865	**760**	**+13.82**
ChatGPT (GPT-3.5)	2	J	52.97	656	500	738	–32.25
Code Llama (Llama 2)	1	P	64.87	492	406	**352**	**+15.34**
Bing AI Chat (GPT-4.0)	3	J	49.05	1910	1846	2048	−9.86
InstructCodeT5+	3	C	56.69	757	829	**650**	**+27.54**
Code Llama (Llama 2)	4	J	53.78	538	574	**498**	**+15.26**
InstructCodeT5+	2	C	64.83	187	356	**279**	**+27.60**
StarCoder	2	P	63.38	254	209	**192**	**+8.85**
ChatGPT (GPT-3.5)	1	P	56.59	697	577	655	−11.91
CodeWhisperer	2	J	57.89	459	355	438	−18.95
Code Llama (Llama 2)	3	P	63.98	364	275	294	−6.46
StarCoder	2	C	57.45	458	395	402	−1.74
InstructCodeT5+	2	J	56.99	519	365	546	−33.15
StarCoder	1	P	57.89	682	628	**550**	**+14.18**
Bing AI Chat (GPT-4.0)	4	J	49.22	648	552	765	−27.84
Bing AI Chat (GPT-4.0)	4	P	57.16	450	436	454	−3.96
CodeWhisperer	2	P	63.55	204	209	**122**	**+71.31**
CodeWhisperer	3	J	56.93	605	752	**605**	**+24.30**

5. Conclusions and Future Work

The fast development of AI raises questions about the impact on developers and development tools since with the help of generative AI, program code can be generated automatically. Consequently, the goal of our paper was to answer the question: How efficient is the program code in terms of computational resources? How understandable and maintainable is the program code for humans? To answer those questions, we analyzed the computational resources of AI- and human-generated program code using metrics such as *time* and *space complexity* as well as *runtime* and *memory usage*. Additionally, we evaluated the *maintainability* using metrics such as *lines of code*, *cyclomatic complexity*, *Halstead complexity* and *maintainability index*.

In our experiments, we utilized generative AIs, including ChatGPT (GPT-3.5), Bing AI Chat (GPT-4.0), GH Copilot (GPT-3.0), StarCoder (StarCoderBase), Code Llama (Llama 2), CodeWhisperer, and InstructCodeT5+ (CodeT5+) to generate program code in Java, Python, and C++. The generated program code aimed to solve problems specified on the coding platform *leetcode.com*. We chose six LeetCode problems with varying difficulty, resulting in the generation of 18 program codes. GH Copilot outperformed others by solving 9 out of 18 coding problems (50.0%), while CodeWhisperer failed to solve any coding problem. BingAI Chat provided *correct* program code for seven coding problems (38.9%), while ChatGPT and Code Llama were successful in four coding problems (22.2%). StarCoder and InstructCodeT5+ each solved only one coding problem (5.6%). GH Copilot excelled in addressing our Java and C++ coding problems, while BingAI demonstrated superior performance in resolving Python coding problems. Surprisingly, while ChatGPT produced only four *correct* program codes, it stood out as the sole model capable of delivering a *correct* solution to a coding problem with a difficulty level of *hard*. This unexpected performance should be further investigated, for example by assessing *pass@k* or further coding problems with difficulty level *hard*.

Figure 4 illustrates in a tree structure an overview of all 122 AI-generated *executable* (96.8%), 4 *not executable* (3.2%), 26 *correct* (20.6%), 96 *incorrect* (76.2%) and 76 *not usable* (57.9% + 2.4%) program codes as well as the 24 program codes that can be made *correct with minimal modifications* (18.3% + 0.8%).

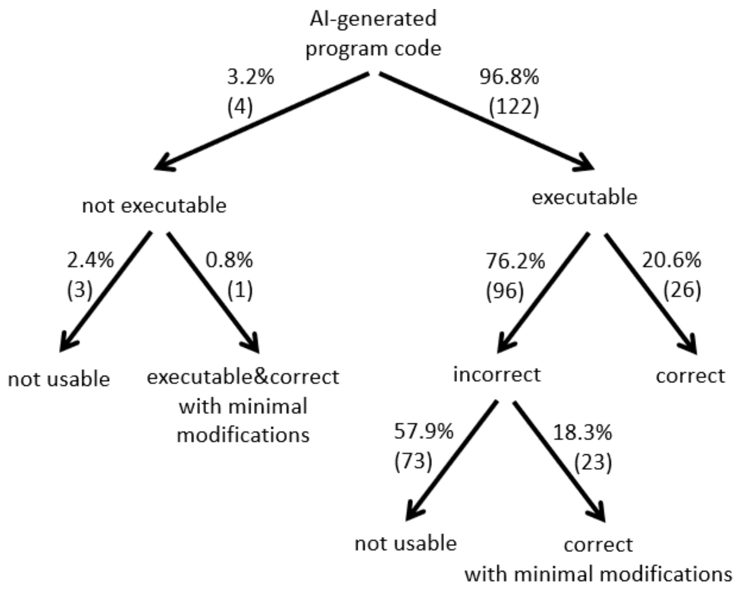


Figure 4. Distribution of AI-generated Codes.

To summarize: We have shown that different state-of-the-art generative AIs perform differently in program code generation depending on the programming language and coding problem. Our experiments demonstrated that we still seem to be a long way from a generative AI that delivers *correct*, *efficient* and *maintainable* program code in every case. However, we have learned that AI-generated program code can have the potential to speed up programming, even if the program code is *incorrect* because often only minor modifications are needed to make it *correct*. For a quick and detailed evaluation of the generated program codes, we used different evaluation metrics and introduced *TTC*, an estimation of the *time to correct incorrect* program code.

In future work, we plan to analyze the quality of AI-generated program code in other programming languages. For that, we will expand our *AI/Human-Generated Program Code Dataset* to cover further programming languages and coding problems. To have a fair comparison among the generative AIs, we applied a prompt engineering strategy that is applicable to a wide range of generative AIs. However, in future work, we plan to investigate the optimal prompting approach for each generative AI individually. Furthermore, we are interested in investigating whether the interaction of different chatbots leveraging different generative AIs helps to improve the final program code quality. For example, as in a human programming team, the generative AIs could take on different roles, e.g., a chatbot that develops the software architecture, a chatbot that is responsible for testing, a chatbot that generates the code or different all-rounders that interact with each other. Since many related works report *pass@k*, we could also have the program codes produced several times for comparability and report *pass@k*. Since the development of generative AIs is rapid, it makes sense to apply our experiments to new generative AIs soon. In this work, we estimated the time for writing and correcting program code based on *Halstead metrics*. But a comparison with the real time required by a representative set of programmers may also be part of future work. We have provided insights into how state-of-the-art generative AI deals with specific coding problems. Moving forward, it would be beneficial to comprehensively investigate how generative AIs handle the generation of more complex programs or even complete software solutions. This could include an analysis of their ability not only to generate intricate algorithms, but also to manage large codebases, use frameworks and libraries in reasonable contexts, and take best practices of software engineering into account. Additionally, it would be interesting to explore how generative AIs could be integrated into existing software development workflows, and whether they could contribute to increased efficiency and productivity.

Author Contributions: Conceptualization, methodology, software, validation, resources, writing, visualization: B.I. and T.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: To contribute to the analysis of generative AIs for program code generation, we share our code and our corpus with the research community: https://github.com/Back3474/AI-Human-Generated-Program-Code-Dataset, accessed on 29 January 2024.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Pelau, C.; Dabija, D.C.; Ene, I. What Makes an AI Device Human-like? The Role of Interaction Quality, Empathy and Perceived Psychological Anthropomorphic Characteristics in the Acceptance of Artificial Intelligence in the Service Industry. *Comput. Hum. Behav.* **2021**, *122*, 106855. [CrossRef]
2. Dibitonto, M.; Leszczynska, K.; Tazzi, F.; Medaglia, C.M. Chatbot in a Campus Environment: Design of LiSA, a Virtual Assistant to Help Students in Their University Life. In *Proceedings of the Human-Computer Interaction*; Interaction Technologies; Kurosu, M., Ed.; Springer: Cham, Switzerland, 2018; pp. 103–116.
3. Arteaga, D.; Arenas, J.J.; Paz, F.; Tupia, M.; Bruzza, M. Design of Information System Architecture for the Recommendation of Tourist Sites in the City of Manta, Ecuador through a Chatbot. In Proceedings of the 2019 14th Iberian Conference on Information Systems and Technologies (CISTI), Coimbra, Portugal, 19–22 June 2019; pp. 1–6.
4. Falala-Séchet, C.; Antoine, L.; Thiriez, I.; Bungener, C. Owlie: A Chatbot that Provides Emotional Support for Coping with Psychological Difficulties. In Proceedings of the 19th ACM International Conference on Intelligent Virtual Agents, Paris, France, 2–5 July 2019.
5. Adiwardana, D.; Luong, M.T.; So, D.R.; Hall, J.; Fiedel, N.; Thoppilan, R.; Yang, Z.; Kulshreshtha, A.; Nemade, G.; Lu, Y.; et al. Towards a Human-like Open-Domain Chatbot. *arXiv* **2020**, arXiv:2001.09977.

6. Schaaff, K.; Reinig, C.; Schlippe, T. Exploring ChatGPT's Empathic Abilities. *arXiv* **2023**, arXiv:2308.03527.
7. Taecharungroj, V. "What Can ChatGPT Do?" Analyzing Early Reactions to the Innovative AI Chatbot on Twitter. *Big Data Cogn. Comput.* **2023**, *7*, 35. [CrossRef]
8. Loh, E. ChatGPT and Generative AI Chatbots: Challenges and Opportunities for Science, Medicine and Medical Leaders. *BMJ Lead.* **2023**. [CrossRef]
9. Mollick, E. ChatGPT Is a Tipping Point for AI. *Harvard Business Review*, 14 December 2022.
10. Alizadehsani, Z.; Gomez, E.G.; Ghaemi, H.; González, S.R.; Jordan, J.; Fernández, A.; Pérez-Lancho, B. Modern Integrated Development Environment (IDEs). In Proceedings of the Sustainable Smart Cities and Territories, Doha, Qatar, 27–29 April 2021; Corchado, J.M., Trabelsi, S., Eds.; Springer: Cham, Switzerland, 2022; pp. 274–288.
11. Kaur, A.; Jadhav, A.; Kaur, M.; Akter, F. Evolution of Software Development Effort and Cost Estimation Techniques: Five Decades Study Using Automated Text Mining Approach. *Math. Probl. Eng.* **2022**, *2022*, 5782587. [CrossRef]
12. Bluemke, I.; Malanowska, A. Software Testing Effort Estimation and Related Problems: A Systematic Literature Review. *ACM Comput. Surv.* **2021**, *54*, 1–38. [CrossRef]
13. Butt, S.A.; Misra, S.; Piñeres-Espitia, G.; Ariza-Colpas, P.; Sharma, M.M. A Cost Estimating Method for Agile Software Development. In Proceedings of the Computational Science and Its Applications— ICCSA 2021, Cagliari, Italy, 13–16 September 2021; Gervasi, O., Murgante, B., Misra, S., Garau, C., Blečić, I., Taniar, D., Apduhan, B.O., Rocha, A.M.A.C., Tarantino, E., Torre, C.M., Eds.; Springer: Cham, Switzerland, 2021; pp. 231–245.
14. Zhang, B.; Liang, P.; Zhou, X.; Ahmad, A.; Waseem, M. Practices and Challenges of Using GitHub Copilot: An Empirical Study. In Proceedings of the International Conferences on Software Engineering and Knowledge Engineering, San Francisco, CA, USA, 1–10 July 2023; KSIR Virtual Conference Center, USA, 2023. [CrossRef]
15. Liu, J.; Xia, C.S.; Wang, Y.; Zhang, L. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. *arXiv* **2023**, arXiv:2305.01210v3.
16. Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; de Oliveira Pinto, H.P.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; et al. Evaluating Large Language Models Trained on Code. *arXiv* **2021**, arXiv:2107.03374.
17. Yetiştiren, B.; Özsoy, I.; Ayerdem, M.; Tüzün, E. Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT. *arXiv* **2023**, arXiv:2304.10778.
18. Wang, B.; Komatsuzaki, A. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model. 2021. Available online: https://github.com/kingoflolz/mesh-transformer-jax/?tab=readme-ov-file#gpt-j-6b (accessed on 29 January 2024).
19. Yetistiren, B.; Ozsoy, I.; Tuzun, E. Assessing the Quality of GitHub Copilot's Code Generation. In Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering, Singapore, 17 November 2022.
20. OpenAI. GPT-4 Technical Report. *arXiv* **2023**, arXiv:2303.08774.
21. Phind. 2023. Available online: https://huggingface.co/Phind/Phind-CodeLlama-34B-v2 (accessed on 12 November 2023).
22. Luo, Z.; Xu, C.; Zhao, P.; Sun, Q.; Geng, X.; Hu, W.; Tao, C.; Ma, J.; Lin, Q.; Jiang, D. WizardCoder: Empowering Code Large Language Models with Evol-Instruct. *arXiv* **2023**, arXiv:2306.08568.
23. OpenAI. Introducing ChatGPT. 2022. Available online: https://openai.com/blog/chatgpt (accessed on 30 September 2023).
24. Rozière, B.; Gehring, J.; Gloeckle, F.; Sootla, S.; Gat, I.; Tan, X.E.; Adi, Y.; Liu, J.; Remez, T.; Rapin, J.; et al. Code Llama: Open Foundation Models for Code. *arXiv* **2023**, arXiv:2308.12950.
25. Li, R.; Ben Allal, L.; Zi, Y.; Muennighoff, N.; Kocetkov, D.; Mou, C.; Marone, M.; Akiki, C.; Li, J.; Chim, J.; et al. StarCoder: May the Source be with You! *arXiv* **2023**, arXiv:2305.06161.
26. Nijkamp, E.; Pang, B.; Hayashi, H.; Tu, L.; Wang, H.; Zhou, Y.; Savarese, S.; Xiong, C. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. *arXiv* **2023**, arXiv:2203.13474.
27. Wang, Y.; Le, H.; Gotmare, A.D.; Bui, N.D.; Li, J.; Hoi, S.C. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. *arXiv* **2023**, arXiv:2305.07922. [CrossRef]
28. Jiang, A.Q.; Sablayrolles, A.; Mensch, A.; Bamford, C.; Chaplot, D.S.; de las Casas, D.; Bressand, F.; Lengyel, G.; Lample, G.; Saulnier, L.; et al. Mistral 7B. *arXiv* **2023**, arXiv:2310.06825.
29. Nijkamp, E.; Hayashi, H.; Xiong, C.; Savarese, S.; Zhou, Y. CodeGen2: Lessons for Training LLMs on Programming and Natural Languages. *arXiv* **2023**, arXiv:2305.02309.
30. Chiang, W.L.; Li, Z.; Lin, Z.; Sheng, Y.; Wu, Z.; Zhang, H.; Zheng, L.; Zhuang, S.; Zhuang, Y.; Gonzalez, J.E.; et al. Vicuna: An Open-Source Chatbot Impressing GPT-4 with 90%* ChatGPT Quality. 2023. Available online: https://lmsys.org/blog/2023-03-30-vicuna (accessed on 29 January 2024).
31. Allal, L.B.; Li, R.; Kocetkov, D.; Mou, C.; Akiki, C.; Ferrandis, C.M.; Muennighoff, N.; Mishra, M.; Gu, A.; Dey, M.; et al. SantaCoder: Don't reach for the stars! *arXiv* **2023**, arXiv:2301.03988
32. Fried, D.; Aghajanyan, A.; Lin, J.; Wang, S.; Wallace, E.; Shi, F.; Zhong, R.; tau Yih, W.; Zettlemoyer, L.; Lewis, M. InCoder: A Generative Model for Code Infilling and Synthesis. *arXiv* **2023**, arXiv:2204.05999.
33. Wang, B. Mesh-Transformer-JAX: Model-Parallel Implementation of Transformer Language Model with JAX. 2021. Available online: https://github.com/kingoflolz/mesh-transformer-jax (accessed on 29 January 2024).
34. Black, S.; Gao, L.; Wang, P.; Leahy, C.; Biderman, S.R. *GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow*; Zenodo; 2021. [CrossRef]

35. Xu, F.F.; Alon, U.; Neubig, G.; Hellendoorn, V.J. A Systematic Evaluation of Large Language Models of Code. In Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming (MAPS 2022), New York, NY, USA, 13 June 2022; pp. 1–10. [CrossRef]
36. Stability-AI. StableLM: Stability AI Language Models. 2023. Available online: https://github.com/Stability-AI/StableLM (accessed on 12 November 2023).
37. Li, Y.; Choi, D.; Chung, J.; Kushman, N.; Schrittwieser, J.; Leblond, R.; Eccles, T.; Keeling, J.; Gimeno, F.; Lago, A.D.; et al. Competition-Level Code Generation with AlphaCode. *Science* **2022**, *378*, 1092–1097. [CrossRef]
38. Nguyen, N.; Nadi, S. An Empirical Evaluation of GitHub Copilot's Code Suggestions. In Proceedings of the 2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR), Pittsburgh, PA, USA, 23–24 May 2022; pp. 1–5. [CrossRef]
39. OpenGenus IQ. GPT-3.5 Model Architecture. 2023. Available online: https://iq.opengenus.org/gpt-3-5-model/ (accessed on 30 September 2023).
40. Choudhry, S. Languages Supported by ChatGPT and How to Use It in Other Languages. 2023. Available online: https://www.mlyearning.org/languages-supported-by-chatgpt/ (accessed on 30 September 2023).
41. Patel, D.; Wong, G. GPT-4 Architecture, Infrastructure, Training Dataset, Costs, Vision, MoE. 2023. Available online: https://github.com/llv22/gpt4_essay/blob/master/GPT-4-4.JPG (accessed on 30 September 2023).
42. Yalalov, D.; Myakin, D. GPT-4's Leaked Details Shed Light on its Massive Scale and Impressive Architecture. *Metaverse Post*, 11 July 2023. Available online: https://mpost.io/gpt-4s-leaked-details-shed-light-on-its-massive-scale-and-impressive-architecture (accessed 29 January 2024).
43. OpenAI. GPT-4. OpenAI Research. 2023. Available online: https://openai.com/gpt-4 (accessed 29 January 2024).
44. GitHub. GitHub Copilot. 2021. Available online: https://github.com/features/copilot/ (accessed on 2 October 2023).
45. Zaremba, W.; Brockman, G. OpenAI Codex. 2021. Available online: https://openai.com/blog/openai-codex/ (accessed on 2 October 2023).
46. Brown, T.B.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; et al. Language Models are Few-Shot Learners. *arXiv* **2020**, arXiv:2005.14165.
47. Hugging Face. llm-Vscode. 2023. Available online: https://marketplace.visualstudio.com/items?itemName=HuggingFace.huggingface-vscode (accessed on 2 October 2023).
48. Phillips, J. StarCoder. 2023. Available online: https://plugins.jetbrains.com/plugin/22090-starcoder/versions (accessed on 2 October 2023).
49. Amazon Web Services, Inc. Amazon CodeWhisperer FAQs. 2023. Available online: https://aws.amazon.com/de/codewhisperer/faqs/ (accessed on 3 October 2023).
50. Amazon Web Services, Inc. CodeWhisperer User Guide. 2023. Available online: https://docs.aws.amazon.com/pdfs/codewhisperer/latest/userguide/user-guide.pdf (accessed on 3 October 2023).
51. Hugging Face. Dataset Card for CodeSearchNet Corpus. 2023. Available online: https://huggingface.co/datasets/code_search_net (accessed on 3 October 2023).
52. Hugging Face. GitHub Code Dataset. 2023. Available online: https://huggingface.co/datasets/codeparrot/github-code (accessed on 3 October 2023).
53. Chaudhary, S. Code Alpaca: An Instruction-following LLaMA Model Trained on Code Generation Instructions. 2023. Available online: https://github.com/sahil280114/codealpaca (accessed on 3 October 2023).
54. LeetCode. LeetCode QuickStart Guide. 2023. Available online: https://support.leetcode.com/hc/en-us/articles/360012067053-LeetCode-QuickStart-Guide (accessed on 10 October 2023).
55. McCabe, T. A Complexity Measure. *IEEE Trans. Softw. Eng.* **1976**, *SE-2*, 308–320. [CrossRef]
56. Cormen, T.; Leiserson, C.; Rivest, R.; Stein, C. *Introduction to Algorithms*, 4th ed.; MIT Press: Cambridge, MA, USA, 2022.
57. Baeldung. Understanding Space Complexity. *Baeldung Comput. Sci.* **2021**. Available online: https://www.baeldung.com/cs/time-vs-space-complexity (accessed on 29 January 2024).
58. Halstead, M.H. *Elements of Software Science*; Elsevier: Amsterdam, The Netherlands, 1977; pp. xiv, 127.
59. Heričko, T.; Šumak, B. Exploring Maintainability Index Variants for Software Maintainability Measurement in Object-Oriented Systems. *Appl. Sci.* **2023**, *13*, 2972. [CrossRef]
60. Microsoft. Visual Studio—Maintainability Index. 2021. Available online: https://docs.microsoft.com/en-us/visualstudio/code-quality/code-metrics-maintainability-index-range-and-meaning (accessed on 27 November 2023).