Evaluating the Performance of LLM-Generated Code for ChatGPT-4 and AutoGen Along with Top-Rated Human Solutions

Ashraf Elnashar, Max Moundas, Douglas C. Schmidt, Jesse Spencer-Smith, Jules White
{ashraf.elnashar, maximillian.r.moundas, d.schmidt, jesse.spencer-smith, jules.white}@vanderbilt.edu
Department of Computer Science, Vanderbilt University, Nashville, Tennessee, USA

Abstract

In the domain of software development, making informed decisions about the utilization of large language models (LLMs) requires a thorough examination of their advantages, disadvantages, and associated risks. This paper provides several contributions to such analyses. It first conducts a comparative analysis, pitting the best-performing code solutions selected from a pool of 100 generated by ChatGPT-4 against the highest-rated human-produced code on Stack Overflow. Our findings reveal that, across a spectrum of problems we examined, choosing from ChatGPT-4's top 100 solutions proves competitive with or superior to the best human solutions on Stack Overflow.

We next delve into the AutoGen framework, which harnesses multiple LLM-based agents that collaborate to tackle tasks. We employ prompt engineering to dynamically generate test cases for 50 common computer science problems, both evaluating the solution robustness of AutoGen vs ChatGPT-4 and showcasing AutoGen's effectiveness in challenging tasks and ChatGPT-4's proficiency in basic scenarios. Our findings demonstrate the suitability of generative AI in computer science education and underscore the subtleties of their problem-solving capabilities and their potential impact on the evolution of educational technology and pedagogical practices.

Keywords: Large Language Models (LLMs), Automated Code Generation, ChatGPT-4 vs. AutoGen Performance, Software Development Efficiency, Stack Overflow Solution Analysis, Computer Science Education, Prompt Engineering in AI Code, Quality Assessment, Runtime Performance Benchmarking, Dynamic Testing Environments.

1 Introduction

Emerging trends, challenges, and research foci. Large language models (LLMs) (5), such as ChatGPT (4) and Copilot (git), have the ability to generate complex code to meet a set of natural language requirements (7). Software developers can use LLMs to generate human descriptions of desired functionality or requirements, as well as synthesize code in a variety of languages ranging from Python to Java to Clojure. These tools are currently being integrated into popular Integrated Development Environments (IDEs), such as IntelliJ (15) and Visual Studio.

LLMs are now easily accessible through the Internet and within IDEs, and developers are increasingly leveraging them to guide many programming tasks. In many cases, the questions and code samples to which developers apply these LLMs are the same questions and code samples they previously would have sought help on via discussion forums. For example, Stack Overflow (`stackoverflow.com`) is a popular online forum where developers ask questions and obtain guidance on code samples.

There has been significant discussion and research (git; 3; 18) on applying LLMs to generate code with respect to the quality of the code from a security and defect perspective. First-generation LLM-based tools often produced poor quality code due to their ability to "hallucinate" convincing text or code that was fundamentally flawed, although it appeared correct. In addition, LLMs trained on human-produced code in open-source projects often had vulnerabilities or eschewed best practices. Much discussion on the code quality generated by LLMs has therefore focused on functional correctness and security.

Although using LLMs before fully comprehending their capabilities and limitations is risky, there are also clear productivity benefits for developers in certain areas. For example, LLMs can help to automate repetitive, tedious, or boring coding tasks and perform these tasks faster—and often better—than developers (9). This productivity boost is particularly apparent when coding tasks involve APIs or algorithms that developers are unfamiliar with and thus require study to master before performing the tasks. When these APIs and algorithms are included in an LLM's training set it often generates code for them swiftly and accurately.

In addition, a key benefit related to code performance is how to employ LLMs via *prompting* and *prompt engineering* for many different potential solutions and then automatically benchmark them to identify the fastest solution(s). A prompt is the natural language input to an LLM (16). Prompt engineering is an emerging discipline that structures interactions with LLM-based computational systems to solve complex problems via natural language interfaces (13).

This paper expands upon our prior work (Elnashar et al.) that compared the runtime performance of code produced by humans vs. code generated by ChatGPT-3.5. We first replicate our earlier experiments replacing ChatGPT-3.5 with ChatGPT-4 (11), which is a more advanced version of the GPT model. As shown below, ChatGPT-4 demonstrates a marked improvement in understanding complex problem statements and generating more efficient code due to its enhanced training data and refined algorithms, which interpret prompts more accurately and increase generated code efficiency.

We next conduct a comparative analysis of AutoGen (19) and ChatGPT-4, revealing notable differences in their success rates and error handling capabilities. In particular, our results reveal that ChatGPT-4's solutions present a 9.8% failure rate and a 90.2% pass rate, whereas AutoGen's solutions have a 15.6% failure rate and an 84.4% pass rate. Moreover, we apply visual tools for clarity and present insights into the potential educational applications of each approach.

Paper organization. The remainder of this paper is organized as follows: Section 2 summarizes the open research questions we address and outlines our technical approach; Section 3 explains our testbed environment configuration and analyzes results from experiments that compare the top Stack Overflow coding solutions against solutions generated by ChatGPT-4; Section 4 examines the effectiveness of the AutoGen approach in generating programming solutions and compares its performance with ChatGPT-4; Section 5 compares our work with related research; and Section 6 presents the lessons learned from our study and outlines future work.

2 Summary of Open Research Questions and Technical Approach

This section summarizes the open research questions we address in this paper and outlines our technical approach for each question.

Q1: How do the most efficient LLM-generated codes from GPT-3.5 Turbo and GPT-4 compare with the top human-produced code in terms of runtime performance? Section 3 investigates the runtime performance of code generated by both GPT-3.5 Turbo and GPT-4, contrasting it with human-produced code. Our analysis includes a comparison of human-written Stack Overflow solutions to those generated by ChatGPT-4 and GPT-3.5 Turbo using diverse prompting strategies. We focus on the efficiency of the fastest solutions from both LLMs compared to the best human answers, representing a real-world scenario where developers might seek the most efficient solution through iterative LLM querying. This investigation provides a foundational understanding of LLMs' utility in practical coding applications.

Q2: What is the range and reliability of coding solutions generated by GPT-3.5 Turbo and GPT-4, compared to a diverse set of human-produced code, in terms of runtime efficiency and practical application? Section 3 expands the scope of our analysis beyond optimal solutions, examining the runtime efficiency of the most common, as well as the best and worst, LLM-generated codes. This study offers a comprehensive view of the coding efficiency that GPT-3.5 Turbo and GPT-4 can achieve, benchmarked against human solutions. By analyzing a range of LLM-generated solutions, we provide insights into the variability and reliability of LLMs as coding assistants.

Q3: Against which human-produced solutions should LLM outputs from GPT-3.5 Turbo and GPT-4 be benchmarked, and what represents the average developer's capability? Section 3 tackles the challenge of setting appropriate benchmarks for LLM-generated code by selecting a representative sample of human solutions for comparison. This analysis helps determine where GPT-3.5 Turbo and GPT-4 stand in relation to average developer skill levels. The chosen benchmarks range from highly optimized to average human solutions, offering a balanced perspective on LLMs' capabilities.

Q4: How does AutoGen, with its systematic and structured LLM prompting, compare with the more flexible and generalized approach of ChatGPT-4 in terms of efficiency, accuracy, and adaptability in code generation? Section 4 expands upon the experiments in Section 3 to assess whether AutoGen's structured prompting leads to more efficient and/or accurate code outputs compared to ChatGPT-4. We apply both AutoGen *and* ChatGPT-4 to evaluate these LLMs' capabilities in comprehending and producing Python code, by presenting them with a sequence of increasingly complex problems. Each generated solution underwent thorough testing for both correctness and efficiency, thus highlighting the LLMs' flexibility and accuracy in code generation.

When addressing these questions, we consider various factors, such as the stochastic nature of LLMs, that may yield different outputs for the same prompt. We also consider the variance in human-provided coding solutions in terms of quality and efficiency. The comparison between AutoGen and ChatGPT-4 further extends this investigation by analyzing the impact of different technical approaches on the quality of the generated code.

Our prior work (21) shows how prompt wording influences the quality of LLM output. We therefore focus on how prompt wording influences the quality of generated code. In particular, we investigate if varying the wording causes LLMs to generate faster code more consistently.

3 Comparing Stack Overview and ChatGPT-4-generated Solutions

This section analyzes the results from our comparison of top human-provided Stack Overflow coding solutions and the corresponding ChatGPT-4-generated solutions.

3.1 Experiment Configuration

This section explains the configuration of our testbed environment and analyzes the results from experiments that compare the top Stack Overflow coding solutions against solutions generated by ChatGPT-4.

3.1.1 Overview of Our Approach

Our analysis was conducted on code samples written in Python since (1) it is relatively easy to extract and experiment with stand-alone code samples in Python compared to other languages, (2) ChatGPT-4 appears to generate more correct code in Python vs. less popular languages (such as Clojure), and (3) Python is a popular language in domains like Data Science where developers often have more familiarity and comfort with LLMs.

Our problem set was manually curated from Stack Overflow by browsing questions related to Python. We searched for questions pertaining to categories, such as "array questions" since these questions are readily tested for performance at increasing input sizes. We then analyzed each question and its candidate solutions to select question/solution pairs that could be isolated and inserted into our test harness.

We avoided questions that relied heavily on third-party libraries to minimize complexity, such as version discrepancies and dependency issues. These complexities can obscure the assessment of the core algorithmic efficiency of the code (a potential threat to validity, as discussed in Section 3.3). Instead, we focused on solutions built on core libraries and capabilities within Python itself.

Wherever possible, we selected the top-voted solution as the comparison. In some cases, multiple languages were present in the solutions and we selected the first Python solution, mimicking developers looking for the first solution in their target language. These decisions and related methodological considerations are discussed further in Section 3.3.4.

For each selected question, we extracted the question's title posted on Stack Overflow and used it as a prompt for ChatGPT-4, leveraging OpenAI's ChatGPT-4 API for this process. This API allowed us to automate sending prompts and receiving code responses, thereby facilitating a consistent and efficient analysis of the model's code generation capabilities. This decision meant that ChatGPT-4 was not provided the full information in the question, which may handicap it in providing better performing solutions. Our rationale for only using question titles as prompts for ChatGPT-4 both reflects common real-world scenarios faced by developers and assesses its ability to generate solutions based on limited information.

The original Stack Overflow posts, human-produced solutions, and ChatGPT-4-generated code solutions—along with our entire set of questions and generated answers—can be accessed in our Github repository at `github.com/elnashara/CodePerformanceComparison`. We encourage readers to replicate our results and submit issues and pull requests for possible improvements.

We measured the runtime performance of each code sample using Python's *timeit* package. Code samples were provided with small, medium, and large inputs. These inputs were progressively increased in size to show the effects of scaling on the generated code. What constituted small, medium, and large was problem-specific, as shown in Section 3.2 below. For each input size, we generated 100 random inputs of the given size to run tests on. In addition, for each input, we tested the given code 100 times on the input using the Python *timeit* package.

3.1.2 Overview of the Coding Problems

A total of 7 problems from Stack Overflow, all pertaining to array operations, were selected for our analysis. These problems encompass a range of array-related challenges, including **PA1**: identifying missing number(s) in an unsorted array, **PA2**: detecting a duplicate number in an array that is not sorted, **PA3**: finding the indices of the k smallest numbers in an unsorted array, **PA4**: counting pairs of elements in an array with a given sum, **PA5**: finding duplicates in a array, **PA6**: removing array duplicates, and **PA7**: implementing the Quicksort algorithm.

3.1.3 Prompting Strategies

In this experiment we applied various prompting strategies to generate Python code with ChatGPT-4, including

1. **Naive approach**, which used only the title from Stack Overflow as the prompt, *e.g.*, "How to count the frequency of the elements in an unordered array",
2. **Ask for speed approach**, which added a requirement for speed at the end of the prompt, *e.g.*, "How to count the frequency of the elements in an unordered array, where the implementation should be fast",
3. **Ask for speed at scale approach**, which provided more detailed information about how the code should be optimized for speed as the size of the array grows, *e.g.*, "How to count the frequency of the elements in an unordered array, where the implementation should be fast as the size of the array grows",

4. **Ask for the most optimal time complexity**, which prioritized achieving the most optimal time complexity, *e.g.*, "How to count the frequency of the elements in an unordered array, where implementation should have the most optimal time complexity possible", and
5. **Ask for the chain-of-thought (23)**, which generated coherent text by providing a series of related prompts, *e.g.*, "Please explain your chain of thought to create a solution to the problem: How to count the frequency of the elements in an unordered array First, explain your chain of thought. Next, provide a step by step description of the algorithm with the best possible time complexity to solve the task. Finally, describe how to implement the algorithm step-by-step in the fastest possible way."

ChatGPT-4 was prompted 100 times with each prompt per coding problem, yielding up to 100 different coding solutions per prompt.[1] We tested the performance of all ChatGPT-4-generated code, however, and did not remove duplicate solutions. If two different prompts had identical solutions, we benchmarked each and left the results with the expectation that 100 timing runs on 100 different inputs would average out any negligible differences in performance.

3.2 Analysis of Experiment Results

The results of our experiment that evaluated the performance of code provided by Stack Overflow and generated by ChatGPT-4 100 times for all seven coding problems with three different input sizes—small (1,000), medium (10,000), and large (100,000)—are shown in Figures 1, 2 and 3. Figure 4 shows the minimum average performance across all input. These figures show the number of problems

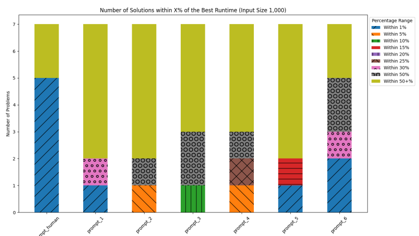

Figure 1. Number of Solutions within X% of the Best Runtime (Input Size 1,000)

for each prompt where the best of the 100 solutions generated by each prompt was within 1%, 5%, etc. of the best solution found across all prompts and the human. For each problem, a total of up to 601 solutions were benchmarked (6 prompts * 100 solutions per prompt + 1 human solution).

[1]In practice, fewer than 100 unique coding solutions were sometimes produced since ChatGPT-4 often generated logically equivalent programs.

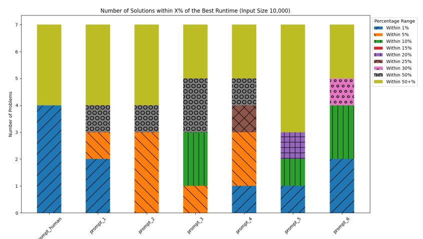

Figure 2. Number of Solutions within X% of the Best Runtime (Input Size 10,000)

The best performing solution was used as the "Best Runtime" solution in the figures against which other solutions were compared. Figures 1, 2, 3 and 4 collectively demon-

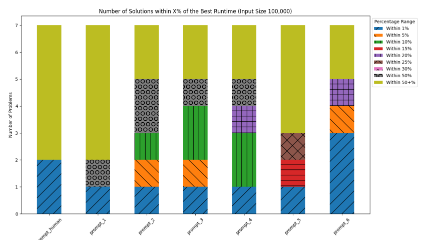

Figure 3. Number of Solutions within X% of the Best Runtime (Input Size 100,000)

strate how ChatGPT-4 selected the best-performing solution out of 100 attempts when employing chain-of-thought reasoning in response to prompts. These solutions were competitive with—and in many cases surpassed—the human-provided solutions from Stack Overflow. This finding is significant as it underscores the potential of LLMs in generating efficient solutions when prompted with a structured approach that includes chain-of-thought reasoning.

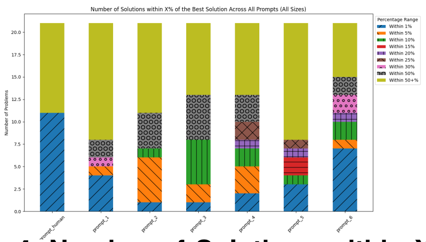

Figure 4. Number of Solutions within X% of the Best Runtime (All Input Sizes)

The human solution was the fastest solution for only one of the problems, specifically the "P2 Find Duplicate Number," as depicted in Figure 5. We used the title of the question as the input to ChatGPT-4. All the code samples produced code with respect to the title of the Stack Overflow post. Since we directly translated the titles into prompts for ChatGPT-4, however, there may have been additional contextual information in the question that ChatGPT-4 could have used to further improve its solution, as discussed in Section 3.3.2.

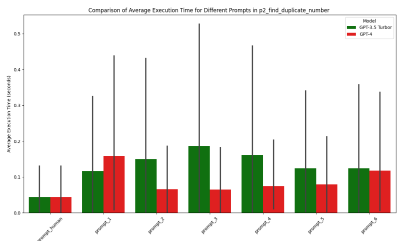

Figure 5. Comparison of Average Execution Time for Different Prompts in P2 Find Duplicate Numbers

Our results also demonstrate a significant improvement in performance when using ChatGPT-4 compared to its predecessor, GPT-3.5 Turbo. This advancement in LLMs underscores the progressive enhancements in AI-driven coding solutions. Despite this progress, the human-crafted solution still outperformed both GPT-4 and GPT-3.5 Turbo for problem P2. This finding suggests that while LLMs are becoming increasingly competent in generating code, there remains an edge that human experience and intuition can provide, particularly in certain complex or nuanced tasks.

Conversely, when evaluating the "P1 Find Missing Number" problem, a distinct change in the hierarchy of solution efficiency was evident. As shown in Figure 6, the human solution was surprisingly the least efficient in terms of execution time, which highlights scenarios where LLMs may exceed human performance. Interestingly, when structured prompt engineering is applied—especially the chain-of-thought method—GPT-3.5 Turbo's capability to devise effective code solutions improves significantly.

In general, however, the most pronounced enhancement is seen with GPT-4, which out-performs both human solutions and GPT-3.5 Turbo when equipped with the same structured prompting techniques. This finding signifies the remarkable advancements in LLMs, especially in the realm of intricate problem-solving. The findings presented in Figure 6 confirm the superior performance of GPT-4 in optimizing code execution time and setting a new threshold in AI-assisted coding (which will likely be surpassed with subsequent releases of ChatGPT).

The contrasting results—with humans prevailing in one case, yet falling behind in another—provides insight into the multifaceted nature of coding solutions within the current LLM landscape. Our research suggests that while LLMs like ChatGPT-4 can outstrip human coders in certain instances, the creativity and specialized skill of human programmers continue to be invaluable assets in complex scenarios. This dynamic highlights the promising potential of a synergistic approach, wherein human expertise is enhanced by the efficiency and evolving capabilities of LLMs,

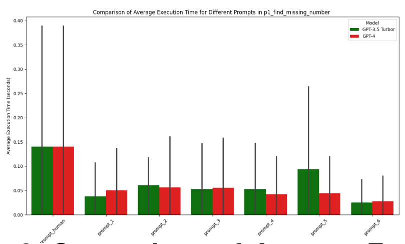

Figure 6. Comparison of Average Execution Time for Different Prompts in P1 Find Missing Number

to elevate the process of developing coding solutions.

3.3 Threats to Validity

Threats to the validity of our experiment results are discussed below.

3.3.1 Sample Size

Although the results presented in Section 3.2 are promising, they are based on a relatively small sample size since our study considered a total of seven computer science (CS) problems, each subjected to 100 testing iterations. While this number of problems and iterations was sufficient to demonstrate initial trends, it does not capture the performance characteristics and potential edge cases encountered in larger datasets. More work on a larger sample size is therefore needed to increase the robustness of our findings.

In general, the software engineering and LLM communities will benefit from a large-scale set of benchmarks that associate (1) code needs (expressed as natural language requirements), questions, specifications, and rules with (2) highly optimized human code, as well as associated benchmarks and interfaces. These communities can then apply the benchmarks to measure and validate LLM coding performance over time to ensure research is headed in the right direction regarding the development and use of generative AI tools.

3.3.2 Prompt Construction

The construction of prompts posed an additional threat to validity because it relied solely on the titles of Stack Overflow questions. In particular, incorporating no additional details from question bodies prevented ChatGPT-4 from utilizing further code to inform its responses. We did not want ChatGPT-4 completing/improving fundamentally flawed code. However, this prompt design choice risked depriving ChatGPT-4 of information it could have used to generate better solutions.

3.3.3 Problem Scope

Another risk area was the variety of coding problems we analyzed. The problems were relatively narrow in scope and data structure type. A wider range of problem types is thus needed to ensure hidden risks regarding specific problem structures do not occur. There may be classes of problems that trigger poor performing hallucinations or code structures we are not aware of yet. This risk is particularly problematic when attempting to generalize our results.

3.3.4 Selection Bias

Another threat to validity was the inherent question and code sample selection bias in our study. These questions and answers were selected manually to focus on problems and code samples that could be tested and benchmarked readily. We may therefore have inappropriately influenced the problem types selected and not chosen samples representative of what developers would ask in certain domains.

4 ChatGPT-4 vs. AutoGen: A Comparative Study in Programming Automation

Computer science and its application domains evolve continuously, requiring more efficient and reliable automated systems capable of solving complex problems. This section systematically compares ChatGPT-4 and AutoGen, which are two generative AI-based systems that enable automated problem-solving. Our comparison evaluates the capability of ChatGPT-4 and AutoGen to (1) generate accurate solutions for a set of predefined computer science problems and (2) successfully pass rigorous tests designed to validate the correctness of these solutions.

ChatGPT-4 was developed as part of OpenAI's GPT series and is adept at a wide range of natural language tasks, catering to diverse users from various domains. Its flexibility and interactivity make it suitable for general inquiries, creative writing, and educational support. In contrast, AutoGen excels in automated code generation through structured and systematic prompting methods that harness predefined patterns and algorithms to craft solutions optimized for accuracy, performance, and readability.

4.1 Problem Statement

AutoGen and ChatGPT-4 both support automated problem-solving and algorithm generation. Little research has been conducted, however, to determine their efficiency and accuracy in producing viable solutions under varying conditions and constraints, especially when the tests themselves are dynamically generated as part of the problem-solving process. Addressing this knowledge gap raises a critical question (question Q4 in Section 2): How reliable are AutoGen and ChatGPT-4 when faced with dynamically changing success criteria, particularly when these criteria are crafted through prompt engineering to match the problem's specific nature?'

The study presented in this section aims to fill the current gap regarding the adaptability and precision of AutoGen and ChatGPT-4 in such fluid testing environments. The absence of predefined tests means the evaluation of these systems must account for their ability to interpret problem statements, generate corresponding tests, and produce solutions that satisfy these tests. What is needed, therefore, is a method that assesses the quality of the generated solutions, as well as the appropriateness and thoroughness of the spontaneously created tests.

By addressing these challenges, we provide a nuanced understanding of the capabilities of ChatGPT-4 and AutoGen. We also explore the extent to which these systems can autonomously generate both problems and their corresponding tests, which is becoming common in continuous integration pipelines and automated software development processes (2). The results of our comparative analysis evaluate the potential of these LLM-driven systems to contribute to and enhance the field of automated software testing and development.

4.2 Dataset Overview and Analysis

The dataset under consideration comprises a collection of 50 computer science problems, each characterized by a unique sequence number, a difficulty level (Category), a ProblemType, and a detailed problem statement. These problems are classified into various categories, reflecting different areas of computer science, such as algorithm design, data structures, and computational theory. The problems are categorized by difficulty levels, ranging from easy to more challenging problems.

This dataset includes a broad spectrum of test cases for each problem, ensuring a comprehensive evaluation of skills from basic functionality to intricate scenarios. For example, test cases for 'Calculating the average of an array of numbers' vary in array sizes and types, while 'Graph traversal' problems test diverse graph structures. This method, akin to our previous study on arrays in Section 3, showcases the range of topics in the dataset, from fundamental algorithms like "Binary Search" to advanced techniques like "Depth-First Search."

The analysis of the distribution of computer science problems by type uncovers the wide range of topics encompassed within the dataset. The pie chart shown in Figure 7 depicts the percentage of problems in each type, providing

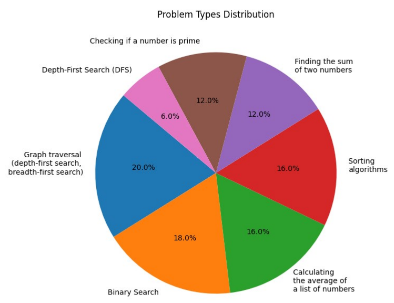

Figure 7. Problem Types Distribution

a visual representation of which areas are emphasized more heavily. This distribution is crucial for understanding the breadth and focus areas of the dataset.

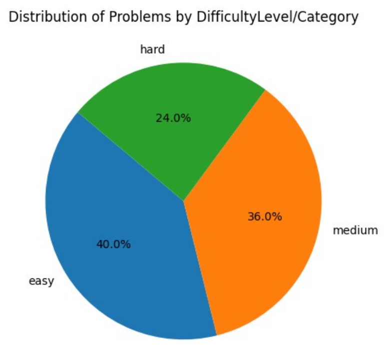

Figure 8. Distribution of Problems by Difficulty Level

The pie chart shown in Figure 8 presents the distribution of problems across different difficulty levels (*i.e.*, easy, medium, and hard) within the dataset. This chart visualizes the proportion of problems in each category, thereby elucidating the distribution pattern. It accentuates the prevalence of specific categories and offers insights into the relative emphasis placed on each difficulty level in our dataset.

4.3 Methodology and Experiment Design

Our experiment design covers the evolving landscape of automated problem-solving and algorithm generation, focusing on the capabilities of ChatGPT-4 and AutoGen. Central to our study is the uniform prompting strategy employed, which is pivotal in harnessing the capabilities of ChatGPT-4 and AutoGen. This strategy applies a consistently structured prompt crafted to convey problem requirements and context uniformly to both AI models. This prompt facilitates a direct comparison of ChatGPT-4 and AutoGen in terms of problem-solving efficiency, accuracy, and adaptability.

By employing this single, standardized prompt across all tests, our study compares and contrasts the performance of these two systems in a controlled and comparable manner. Given the dynamic nature of our problem-solving environment—where tests are not static but generated in response to each unique problem—our study evaluates the efficiency and accuracy of these systems under these varying conditions.

4.3.1 Problem-Solving and Test Generation Approach

Our approach is anchored in prompt engineering (8), which guides LLMs to interpret problem statements and generate corresponding solutions and tests. We give both ChatGPT-4 and AutoGen the same structured prompt shown in Figure 9, which provides the foundation for both systems to understand and approach the problem. This prompt was crafted to outline the problem statement, solution development requirements, script necessities, test case execution and preparation, and execution process. Our approach enables a fair comparison between ChatGPT-4 and AutoGen, ensuring the focus remains on the ability of these systems to generate solutions, as well as create relevant and comprehensive test cases.

```
## Problem Statement
- Develop a Python script to solve the problem: '{Implementing a recursive DFS algorithm to
traverse a binary tree.}'

## Solution Development
- Create a Python function named 'funcImp' that implements the solution.
- Ensure that the function is defined at the beginning of your script and is accessible throughout
the script.

## Script Requirements
- The script should define the 'funcImp' function at the root level, not inside any class or
other function.
- Include comments in the script to explain the logic and functionality of the 'funcImp' function.
- Test the function within the script to ensure it's correctly defined and functioning as expected.

## Test Case Execution
- Execute the 'funcImp' function with various test cases to verify its correctness.
- Ensure that the function 'funcImp' is defined and accessible in the scope where the test cases
are executed.

## Test Case Preparation
- Prepare a set of test cases, including edge cases, to thoroughly test the function.
- Test cases should cover different types of input strings, such as alphabetic, numeric, special
characters, and empty strings.

## Execution Process
- Run each test case through the 'funcImp' function.
- Capture the output of each test case to compare it with the expected result.
```

Figure 9. Structured Prompt Example for LLM-Based Solution Generation in CS Problems.

4.3.2 Evaluating ChatGPT-4 and AutoGen

The evaluation of ChatGPT-4 and AutoGen involved multiple layers. First, we assessed these systems' ability to interpret problem statements accurately and generate viable solutions. Second, we examined the appropriateness and thoroughness of the spontaneously created test cases. These test cases were vital to our evaluation process since they represented the dynamic criteria against which the generated solutions were measured.

Our assessment compared the solutions and tests generated by each system under identical problem conditions. This comparative analysis evaluated the adaptability, precision, and reliability of ChatGPT-4 and AutoGen in a fluid testing environment where both the problems and their corresponding tests were generated autonomously.

This study provided a nuanced understanding of the capabilities of ChatGPT-4 and AutoGen in automated problem-solving and test generation. Our work is particularly pertinent in contexts like continuous integration pipelines and automated software development processes, where the ability to autonomously generate and test solutions is vital. The findings of our study provide insight into the potential role of LLM-based systems in enhancing automated software testing and development.

4.4 Analysis of ChatGPT-4 Experiment Results

The experiment conducted using ChatGPT-4's solution generation capabilities provided a comprehensive view of its performance across a range of computer science problems. To ensure a fair and accurate comparison, the same set of 50 distinct problems, along with identical prompts, were utilized for both ChatGPT-4 and AutoGen in the tests. Figure 10 providing valuable insights into the effectiveness of the generated solutions.

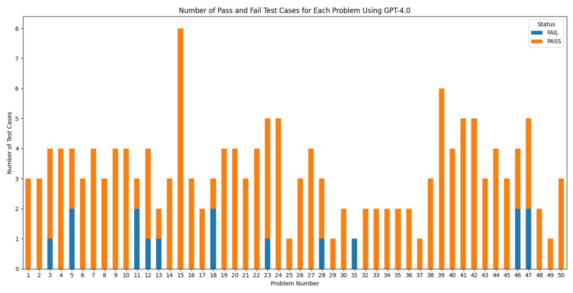

Figure 10. ChatGPT-4 - Pass Rate of Solutions

4.4.1 Overall Success Rate

ChatGPT-4's overall success rate was approximately 90.2%, as shown in Figure 11. This success rate indicates

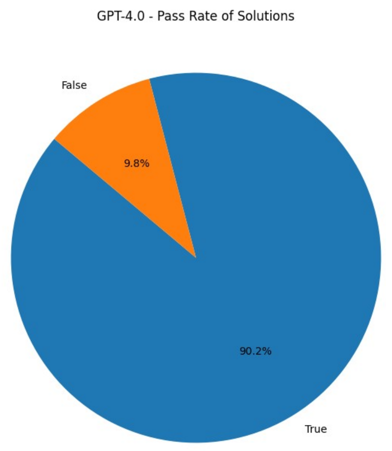

Figure 11. ChatGPT-4 - Pass Rate of Solutions

ChatGPT-4's capability in accurately solving a broad spectrum of computational tasks. The high percentage of correctly solved problems demonstrates the effectiveness of its generated solutions in various contexts.

4.4.2 Error Analysis

Distinct patterns emerged when examining ChatGPT-4's failed cases, highlighting areas where it faced challenges. The most frequent error encountered was related to ”Invalid input. Please provide valid numeric values,” followed by issues like ”max() arg is an empty sequence” and ”division by zero.” These errors indicate that while ChatGPT was proficient in many areas, there were specific scenarios where improvements were needed, particularly involving input validation and handling exceptional cases.

4.4.3 Problem Difficulty vs. Success Rate

An interesting aspect of ChatGPT-4's behavior is the correlation between problem difficulty and success rate. Surprisingly, 'medium' difficulty problems had a higher success rate (around 93.48%) compared to 'easy' (87.50%) and 'hard' (91.11%) difficulties, as shown in Figure 12. This

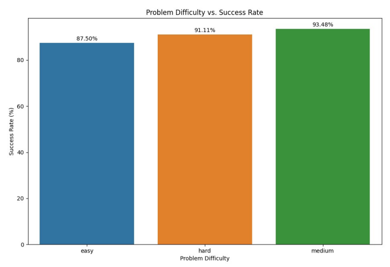

Figure 12. ChatGPT-4 - Problem Difficulty vs. Success Rate

finding suggests a potential discrepancy in the perceived versus actual complexity of the problems or a higher adaptability of the system in solving medium complexity tasks.

4.4.4 Problem Type Analysis

ChatGPT-4's success rate also varied significantly across different problem types. Types such as ”Binary Search” and ”Sorting algorithms” demonstrated a notably high success rate (over 90%), whereas ”Graph traversal” and ”Calculating the average of an array of numbers” exhibited lower success rates. This variation highlighted ChatGPT-4's strengths and weaknesses in different computational domains and offered insights for targeted improvements in specific areas of problem-solving.

4.4.5 Insights and Future Directions

Overall, our analysis of ChatGPT-4's experiment results reveals that it was highly effective in solving a wide range of computer science problems. However, the insights gained from the error analysis and the variation in success rates across problem types and difficulties suggest areas for further enhancement. Improving input validation, error handling, and adapting strategies for specific problem types could yield even higher success rates and more robust problem-solving for ChatGPT-4. These findings help inform future developments to refine the solution generation capabilities of ChatGPT-4.

4.5 Analysis of AutoGen Experiment Results

The experiment conducted on the auto-generation system for computer science problems provided a wealth of data, allowing an in-depth analysis of its performance. The dataset comprises results from tests conducted on 50 different computer science problems shown in Figure 13, where each test was evaluated across multiple parameters.

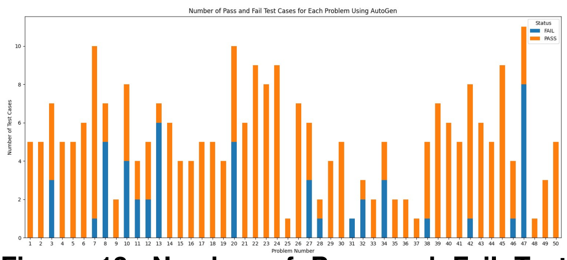

Figure 13. Number of Pass and Fail Test Cases for Each Problem Using AutoGen

4.5.1 Overall Success Rate

AutoGen achieved an overall success rate of approximately 84.35% Figure 14. This high percentage indicates that it

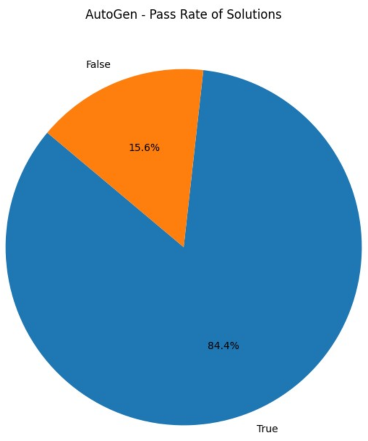

Figure 14. AutoGen - Pass Rate of Solutions

solves the majority of the problems correctly by the auto-generated solutions. It reflects AutoGen's proficiency in handling a range of computational tasks and its effectiveness in producing accurate solutions.

4.5.2 Problem Difficulty vs. Success Rate

Understanding the relationship between problem difficulty and success rate is crucial to assess the effectiveness of solution generation methods. The bar chart shown in Figure 15 visualizes the success rates of solutions across dif-

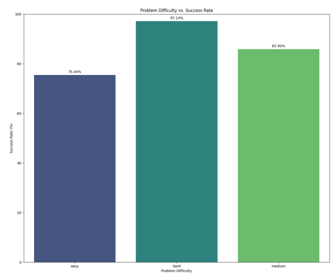

Figure 15. AutoGen - Problem Difficulty vs. Success Rate

ferent problem difficulties in our dataset and distinguishes problem difficulties, such as 'easy', 'medium', and 'hard', represented by individual bars. The height of each bar signifies the percentage of successful solutions within that specific difficulty category. This visualization enables an intuitive comparison of success rates across different levels of problem complexity.

AutoGen's approach, characterized by structured LLM prompting, is highly effective for complex problems, which may account for the lower success rates in 'easy' problems. Its design seems tailored for intricate scenarios requiring deep analysis, leading to better performance in 'medium' and 'hard' problems. This insight helps explain AutoGen's proficiency with complex issues and its less effective handling of simpler tasks.

Contrary to expectations, Figure 15 reveals that 'easy' problems have the lowest success rate, suggesting a mismatch between perceived simplicity and actual solution effectiveness. Conversely, as we move towards 'medium' and 'hard' problems, there is a noticeable increase in success rates, which implies that more complex problems might be better suited to the solution generation and testing processes, leading to higher success rates. The quantification of success in percentages adds precision to our analysis, enabling a more accurate evaluation of solution performance across different problem types.

4.5.3 Failed Cases Analysis

Two distinct patterns were identified in our analysis of failed cases, shedding light on specific challenges faced by AutoGen. One issue occurred in problems dealing with *the calculation of the average of an array of numbers*, where it struggled with handling 'NoneType' values. In particular, AutoGen yielded errors where a floating-point number was expected as a string or a real number, but 'NoneType' was encountered instead.

Another area of difficulty was observed in *sorting algorithms*. In this area AutoGen faced challenges due to string data type limitations, as shown by errors indicating that a string does not support item assignment. This finding indicated potential issues in AutoGen's approach to implementing or understanding the intricacies of sorting strings.

These insights suggest that while AutoGen was largely successful, it can be improved in certain areas. In particular, its handling of edge cases and specific data types requires attention. These patterns can guide future enhancements to AutoGen for better accuracy and robustness in solution generation.

4.6 Comparative Analysis of ChatGPT-4 and AutoGen Experiment Results

Conducting a detailed comparative analysis between the ChatGPT-4 and AutoGen experiment results revealed several key distinctions and similarities. It also offered insightful perspectives on the performance and application of each system. Our analysis begins by examining the overall success rates of both systems, as shown in Figure 16. This figure shows ChatGPT-4 achieved a higher success rate

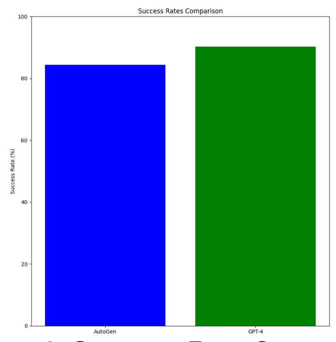

Figure 16. Success Rate Comparison

($sim90.2\%$) indicating its effectiveness in generating correct solutions for the given programming problems. In contrast, AutoGen demonstrated a somewhat lower success rate ($\sim$84.35%), though it is still a substantial majority. This finding suggests that while AutoGen is largely reliable, it may encounter more challenges or inconsistencies in generating correct solutions.

With respect to error analysis, Figure 17 shows the differences become more pronounced. Of the AutoGen tests

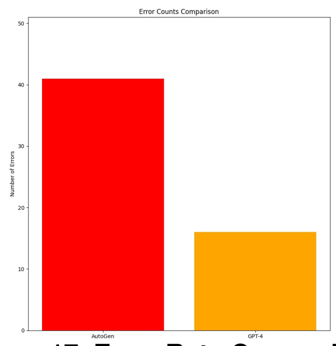

Figure 17. Error Rate Comparison

that did not pass, only two instances recorded specific exceptions. Most of the errors (39 out of 41) lacked detailed exception information, which implied a range of underlying issues, from logic errors to unhandled exceptions in the code.

Conversely, ChatGPT-4 had a lower overall error rate, with 16 instances of failed tests. Notably, ChatGPT-4 documented specific exceptions in one out of these 16 errors. This result offers better insight into the nature of the issues, which included input validation errors and undefined variables.

We also analyzed the complexity of problems and the handling of solutions by both systems. Although tasked with similar problems (primarily basic arithmetic operations), ChatGPT-4's solutions exhibited capabilities for handling more complex scenarios, such as error handling and input validation. Conversely, AutoGen showed a higher error rate and its solutions lacked this complexity in error handling within the sample data.

Summarizing our comparative analysis, both AutoGen and ChatGPT-4 exhibit distinct strengths and limitations in programming solution generation. AutoGen's slightly lower success rate suggests it is most effective for educational use and basic programming tests. Despite ChatGPT-4's higher error rate in complex scenarios, it shows advanced capabilities like robust error handling and input validation, positioning it as a valuable tool for more advanced learning and comprehensive testing environments. These differences highlight the potential applications and suitability of each system in varying contexts of programming education and automated solution testing.

5 Related Work

The evolution of LLMs in code generation has been pivotal, particularly in the discipline of prompt engineering, which focuses on crafting effective natural language inputs for LLMs, enabling them to solve complex problems across diverse domains (8). Studies in this area have emphasized the importance of prompt structure and leveraged external tools and methods to enhance the capabilities of LLMs in coding tasks (22). For instance, Yao et al. (2022) integrated LLMs with external coding frameworks to augment their utility, while Van et al. (2023) focused on maximizing the inherent capabilities of LLMs in generating more complex and efficient code structures (20). These advancements in prompt engineering show particularly promising results in domains like mathematics, where straightforward prompting often falls short, necessitating more sophisticated approaches for better outcomes (12).

Our study delves deeper into the impact of prompt design on the performance of LLM-generated code. Existing research primarily employs direct queries from sources like Stack Overflow, providing a baseline for our investigation. However, the potential for refined prompting techniques to yield more efficient and accurate code solutions suggests an expansive field ripe for future exploration. This area of research is critical, especially considering the increasing reliance on AI-driven solutions in software development.

Moreover, the reliability and security of code generated by LLMs have become focal points in recent studies. Researchers like Borji et al. (2023) and Frieder et al. (2023) have identified and addressed various bugs and security vulnerabilities inherent in LLM-generated code (6; 12). The comparison of the security profile of human-written code versus LLM-generated code, as explored by Asare et al. (2022), is also garnering significant attention (14; 17; 3). This line of research is crucial in understanding the trade-offs between human and AI-generated code, especially concerning security and reliability aspects.

Another emerging area of interest is the impact of LLMs on software development workflows and developer productivity. Studies have begun to assess how LLMs influence the software development lifecycle, from initial design to deployment, and their role in accelerating development processes while maintaining, or even improving, code quality. This aspect is particularly relevant as the industry gravitates towards more AI-integrated development environments.

Overall, the body of research underscores the multifaceted impact of LLMs in programming. It highlights the challenges in ensuring the reliability and security of LLM-generated code while also exploring the opportunities in enhancing the efficiency and effectiveness of software development practices. As LLMs continue to evolve, so too does the landscape of research, continually pushing the boundaries of what can be achieved through AI-driven code generation and opening new frontiers in the intersection of AI and software engineering.

6 Concluding Remarks

This paper presented a comprehensive analysis of programming automation, comparing AutoGen and ChatGPT-4, and evaluating top Stack Overflow solutions against those generated by ChatGPT-4. We observed that ChatGPT-4 can produce solutions competitive with human-crafted ones, especially when guided by chain-of-thought reasoning. This approach enhances its problem-solving and code generation capabilities.

In contrast, despite AutoGen's slightly lower success rate, it excels in handling complex programming challenges with robust error handling and input validation, making it suitable for advanced education and testing. ChatGPT-4, however, demonstrated versatility in generating optimized solutions for various problems when effectively prompted.

Key lessons learned from this research include:

- Our experiments demonstrated that prompting and automatically benchmarking generated code effectively leverages LLMs for optimized code. As shown in Section 3, prompting multiple times and selecting the best solution is a promising aid for software engineers to optimize performance-critical code sections.
- A key attribute of ChatGPT-4-based code generation is its ability to search many coding solutions. Developers will likely use LLM-based tools like Code Inspector and Auto-GPT to generate and analyze multiple solutions per query, as discussed in Section 4. Future tools should enable defining metrics and automatically prompting until a quality threshold is met, a prompt limit is reached, and/or time runs out.
- ChatGPT-4 demonstrated a robust 90.2% success rate, and was particularly effective for simpler arithmetic tasks, making it valuable for education and automated testing. As noted in Section 4.4.2, however, its error diagnosis and reporting need further refinement.
- AutoGen's 84.35% success rate demonstrated advanced solutions featuring error handling and input validation, as described in Section 4.5. This finding indicates AutoGen's suitability for advanced education and comprehensive testing environments where robust error handling is essential

In summary, our analysis of program automation using AutoGen and ChatGPT-4 reveals distinct strengths in each system: AutoGen for basic educational use and straightforward problem-solving and ChatGPT-4 for advanced programming solutions and robust error handling, particularly when utilizing chain-of-thought reasoning.

Moreover, our analysis demonstrated that ChatGPT-4 can generate solutions competitive with—or superior to—top Stack Overflow answers when given effective prompts. This finding highlights the potential of LLMs in complex coding tasks but also points to the limitations of using minimal context from Stack Overflow titles. Optimized prompting strategies are essential to fully leverage LLM capabilities in code generation. The choice between these two systems should therefore be guided by the specific needs of the application, *i.e.*, whether the priority lies in maximizing successful outcomes or in handling complex programming challenges with sophisticated error processing.

An intriguing direction for future work is exploring the potential of leveraging LLM-based tools for full stack software development. Rather than focusing solely on individual modules or components, we plan to investigate how LLMs perform at generating complete end-to-end systems encompassing front-end, back-end, database, and infrastructure elements. Examining the effectiveness of LLMs across the entire software lifecycle may reveal new capabilities and limitations. Key areas of analysis include correctness, security, scalability, maintainability, and modularity of auto-generated systems. In addition, studying integration with human developers in a blended workflow rather than as a wholesale replacement will provide important insights.

Our future work will also consider if/how other code quality metrics can be integrated to allow considering multiple dimensions of code quality beyond performance. In particular, security and functional correctness are clearly important points of consideration, but must be supplemented with additional analyses. Likewise, other quality attributes, such as memory consumption, long-term maintainability, and modularity, should also be analyzed. As LLMs continue to mature, understanding their role in higher-level software creation and complementing human programmers offer promising new frontiers.

Acknowledgements

We used ChatGPT-4's Advanced Data Analysis capabiity to generate code for the data visualizations and filter the data sets.

References

[git] Github copilot · https://github.com/features/copilot.

[2] Arachchi, S. and Perera, I. (2018). Continuous integration and continuous delivery pipeline automation for agile software project management. In *2018 Moratuwa Engineering Research Conference (MERCon)*, pages 156–161.

[3] Asare, O., Nagappan, M., and Asokan, N. (2022). Is github's copilot as bad as humans at introducing vulnerabilities in code? *arXiv preprint arXiv:2204.04741*.

[4] Bang, Y., Cahyawijaya, S., Lee, N., Dai, W., Su, D., Wilie, B., Lovenia, H., Ji, Z., Yu, T., Chung, W., et al. (2023). A multitask, multilingual, multimodal evaluation of chatgpt on reasoning, hallucination, and interactivity. *arXiv preprint arXiv:2302.04023*.

[5] Bommasani, R., Hudson, D. A., Adeli, E., Altman, R., Arora, S., von Arx, S., Bernstein, M. S., Bohg, J., Bosselut, A., Brunskill, E., et al. (2021). On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*.

[6] Borji, A. (2023). A categorical archive of chatgpt failures. *arXiv preprint arXiv:2302.03494*.

[7] Carleton, A., Klein, M. H., Robert, J. E., Harper, E., Cunningham, R. K., de Niz, D., Foreman, J. T., Goodenough, J. B., Herbsleb, J. D., Ozkaya, I., and Schmidt, D. C. (2022). Architecting the future of software engineering. *Computer*, 55(9):89–93.

[8] Chen, B., Zhang, Z., Langrené, N., and Zhu, S. (2023). Unleashing the potential of prompt engineering in large language models: a comprehensive review.

[9] De Vito, G., Lambiase, S., Palomba, F., Ferrucci, F., et al. (2023). Meet c4se: Your new collaborator for software engineering tasks. In *2023 49th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 235–238.

[Elnashar et al.] Elnashar, A., Moundas, M., Schimdt, D. C., Spencer-Smith, J., and White, J. Prompt engineering of chatgpt to improve generated code & runtime performance compared with the top-voted human solutions.

[11] Espejel, J. L., Ettifouri, E. H., Alassan, M. S. Y., Chouham, E. M., and Dahhane, W. (2023). Gpt-3.5, gpt-4, or bard? evaluating llms reasoning ability in zero-shot setting and performance boosting through prompts. *Natural Language Processing Journal*, 5:100032.

[12] Frieder, S., Pinchetti, L., Griffiths, R.-R., Salvatori, T., Lukasiewicz, T., Petersen, P. C., Chevalier, A., and Berner, J. (2023). Mathematical capabilities of chatgpt. *arXiv preprint arXiv:2301.13867*.

[13] Giray, L. (2023). Prompt engineering with chatgpt: A guide for academic writers. *Annals of biomedical engineering*, 51(12):2629—2633.

[14] Jalil, S., Rafi, S., LaToza, T. D., Moran, K., and Lam, W. (2023). Chatgpt and software testing education: Promises & perils. *arXiv preprint arXiv:2302.03287*.

[15] Krochmalski, J. (2014). *IntelliJ IDEA Essentials*. Packt Publishing Ltd.

[16] Liu, P., Yuan, W., Fu, J., Jiang, Z., Hayashi, H., and Neubig, G. (2023). Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Computing Surveys*, 55(9):1–35.

[17] Nair, M., Sadhukhan, R., and Mukhopadhyay, D. (2023). Generating secure hardware using chatgpt resistant to cwes. *Cryptology ePrint Archive*.

[18] Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., and Karri, R. (2022). Asleep at the keyboard? assessing the security of github copilot's code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 754–768. IEEE.

[19] Porsdam Mann, S., Earp, B. D., Møller, N., Vynn, S., and Savulescu, J. (2023). Autogen: A personalized large language model for academic enhancement—ethics and proof of principle. *The American Journal of Bioethics*, 23(10):28–41.

[20] van Dis, E. A., Bollen, J., Zuidema, W., van Rooij, R., and Bockting, C. L. (2023). Chatgpt: five priorities for research. *Nature*, 614(7947):224–226.

[21] White, J., Fu, Q., Hays, S., Sandborn, M., Olea, C., Gilbert, H., Elnashar, A., Spencer-Smith, J., and Schmidt, D. C. (2023). A prompt pattern catalog to enhance prompt engineering with chatgpt. *arXiv preprint arXiv:2302.11382*.

[22] Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., and Cao, Y. (2022). React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*.

[23] Zhang, Z., Zhang, A., Li, M., and Smola, A. (2022). Automatic chain of thought prompting in large language models. *arXiv preprint arXiv:2210.03493*.