Advancing Generative AI in Software Development: Evaluating LLM-Generated Code Against Top 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

Making informed decisions about applying large language models (LLMs) in the domain of software development requires thoroughly examining their advantages, disadvantages, and associated risks. This paper extends our prior work by comparing multiple LLMs, including ChatGPT-4, AutoGen, and the recently released ChatGPT-4o, to evaluate their performance in code generation tasks. We conduct a systematic analysis of the best-performing code solutions generated by these LLMs against the highest-rated human-produced code on Stack Overflow. Our findings reveal that the top solutions generated by ChatGPT-4o are competitive with—or superior to—the best human solutions across a spectrum of problems, whereas AutoGen demonstrates distinct strengths in handling complex tasks.

We next delve deeper into AutoGen, which harnesses multiple LLM-based agents to tackle tasks collaboratively. We employ prompt engineering to generate test cases dynamically for 50 common computer science problems and then (1) analyze the solution robustness of Autogen vs. ChatGPT-4o and (2) quantify AutoGen's effectiveness in complex tasks and ChatGPT-4o's proficiency in basic tasks. Our findings evaluate the suitability of LLMs in computer science education and underscore their problem-solving capabilities, as well as their potential impact on the evolution of educational technology and pedagogical practices.

Keywords: LLMs, Automated Code Generation, Performance of ChatGPT-4o vs. AutoGen, Software Development Efficiency, Prompt Engineering.

1 Introduction

Emerging trends, challenges, and research foci. Large language models (LLMs) [4], such as ChatGPT [3], can generate complex code to meet a range of natural language requirements [6]. Software developers use LLMs to generate descriptions of desired functionality or requirements, as well as synthesize code in a variety of languages ranging from Python to Java and Clojure. These tools are being integrated into popular Integrated Development Environments (IDEs), such as IntelliJ [16] and Visual Studio.

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

Significant research and development has focused on applying LLMs to generate quality code from a security and defect perspective. First-generation LLMs (such as ChatGPT-3.5) often produced poor quality code due to their tendency to "hallucinate" convincing text or code that was fundamentally flawed, although it appeared correct [14]. In addition, LLMs trained on human-produced code in open-source projects initially incurred vulnerabilities or eschewed best practices [2]. Much research on the code quality generated by LLMs has thus focused on functional correctness and security [20].

Although it is risky to use LLMs before fully comprehending their capabilities and limitations, developers obtain clear productivity benefits in certain areas. For example, LLMs help automate repetitive and tedious coding tasks and perform these tasks faster—and often better—than developers [8]. This productivity boost is particularly evident when coding tasks involve APIs or algorithms that developers are unfamiliar with and thus require detailed study to master before performing the tasks. Moreover, when these APIs and algorithms are included in an LLM's training set it often generates code for them swiftly and accurately [11].

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 benchmark them dynamically to identify the most efficient solution(s). A prompt is the natural language input to an LLM [17, 26]. Prompt engineering is an emerging discipline that structures interactions with LLM-based computational systems to solve complex problems via natural language interfaces [13, 25].

This paper expands our prior work [9] that compared the runtime performance of code produced by humans vs. code generated by ChatGPT-4 in the following ways:

- We extend our earlier experiments by replacing ChatGPT-4 with the recent ChatGPT-4o [10] release, which consistently performed better in our experiments. In particular, ChatGPT-4o demonstrated a marked improvement in understanding complex problem statements and generating more efficient code. These improvements are attributed to ChatGPT-4o's enhanced training data and refined algorithms, which enable it to interpret prompts more accurately and generate more efficient code.
- We also incorporated ChatGPT-4o into our comparison of AutoGen [22] and the ChatGPT family of LLMs. This addition offers insights into how these AI tools differ in terms of accuracy, scalability, and their ability to handle diverse programming challenges. In particular, our comparative analysis of ChatGPT-4o and AutoGen show they exhibit slightly different success rates and error handling capabilities, *e.g.*, ChatGPT-4o's solutions achieve a (92.8%) pass rate with a (7.2%) failure rate, while AutoGen's solutions reach a (93.6%) pass rate and a (6.4%) failure rate.
- We broadened the scope of our analysis by incorporating new coding problem domains and datasets, assessing how both AI tools perform with larger and more complex tasks. This scalability analysis provides new comparative data, highlighting performance trends across a range of input sizes and complexities.
- A thorough analysis of specific errors encountered by each AI tool is provided, including their root causes and potential mitigations. This analysis enables a more nuanced understanding of the strengths and weaknesses of each AI tool than our prior work.
- Our prior work [25] showed how prompt wording influences the quality of LLM output. We extend that analysis by focusing on how prompt wording affects not only code quality but also runtime performance and scalability. In particular, we investigate whether varying prompt wording consistently produces faster and more efficient code across different LLMs.

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 and analyzes results from experiments that compare the top Stack Overflow coding solutions against solutions generated by ChatGPT-4o; Section 4 examines the effectiveness of applying AutoGen to generate programming solutions and compares its performance with ChatGPT-4o; Section 5 compares our research with related work; and Section 6 presents 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 the technical approach applied to each question.

Q1: How does the most efficient LLM-generated code from GPT-4o compare with the top human-produced code in terms of runtime performance? This question is addressed in Section 3, particularly Sections 3.2 (Analysis of Experiment Results) and 3.3 (Threats to Validity). Section 3.2 provides a detailed analysis of the runtime performance of code generated by GPT-4o and human-produced solutions, including comparing their performance across different input sizes and coding tasks. Likewise, Section 3.3 discusses potential factors that influence runtime performance results, including the limitations in sample size and problem selection.

Q2: What is the range and reliability of code generated by GPT-4o in terms of runtime efficiency and practical application compared to a diverse set of human-produced code? This question is explored in Section 3, specifically Sections 3.1.3 (Prompting Strategies) and 3.2 (Analysis of Experiment Results). Section 3.1.3 examines how different prompting techniques affect the range of solutions generated and their applicability in real-world scenarios, providing a more refined view of LLM performance. Section 3.2 analyzes the variability and reliability of GPT-4o solutions across a broad spectrum of tasks.

Q3: Against which human-produced solutions should LLM outputs from GPT-4o be benchmarked and what represents the "average" developer's capability? This question is addressed in Section 3, specifically Sections 3.1.1 (Overview of Our Approach) and 3.1.2 (Overview of the Coding Problems). Section 3.1.1 outlines the method used to select representative human-produced solutions, focusing on selecting code samples that reflect a broad range of developer skills, from highly optimized solutions to those of average programmers. Section 3.1.2 describes the coding problems chosen for benchmarking, ensuring that the selected human solutions are appropriate for comparing LLM outputs to software developer capabilities.

Q4: How does AutoGen, with its systematic and structured LLM prompting, compare with the more flexible and generalized approach of ChatGPT-4o in terms of efficiency, accuracy, and adaptability in code generation? We explore this question in Section 4, specifically Sections 4.1 (Problem Statement) and 4.3 (Methodology and Experiment Design). Section 4.1 provides the context and goals of comparing ChatGPT-4o and Autogen, emphasizing their different approaches to code generation. Section 4.3 outlines our experiment design, explaining how systematic prompting in AutoGen contrasts with ChatGPT-4o's more flexible approach, particularly for tests on in-

creasingly complex problems. The results assess both efficiency and accuracy in code outputs.

When addressing these four questions, we consider factors like the stochastic nature of LLMs and the variance in human-provided coding solutions with respect to quality and efficiency. This paper also introduces a new factor—scalability of LLMs—that explores how well these AI tools perform as problem complexity increases, which was absent in our previous work. The comparison between ChatGPT-4o and Autogen in Section 4 further extends this investigation by analyzing the impact of different technical approaches on the quality of generated code.

3 Comparing Stack Overview and ChatGPT-4o-generated Solutions

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

3.1 Experiment Configuration

Below we explain the configuration of our testbed environment and analyze the results from experiments that compare the top Stack Overflow coding solutions against solutions generated by ChatGPT-4o.

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, (2) ChatGPT-4o generates better code in Python than in less common languages like Clojure, and (3) Python is a popular language in domains like Data Science where developers are often familiar with LLMs.

We manually curated our problem set from Stack Overflow by browsing questions related to Python. We searched for categories like "array questions" since they are readily testable for performance at increasing input sizes. We next analyzed each question and its candidate solutions to select question/solution pairs that were isolated and inserted into our test harness (see Section 3.2).

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

Wherever possible, we selected the top-voted solution for comparison. In some cases, multiple programming languages were present in the solutions, so we selected the first Python solution, mimicking developers looking for the first solution in their target language. These decisions and related methodology considerations are covered 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-4o, leveraging OpenAI's ChatGPT-4o API for this process. This API automated sending prompts and receiving code responses, thereby facilitating a consistent and efficient analysis of its code generation capabilities. This decision meant that ChatGPT-4o was not provided the full information in the question, however, which may handicap it by providing less optimial solutions.[1]

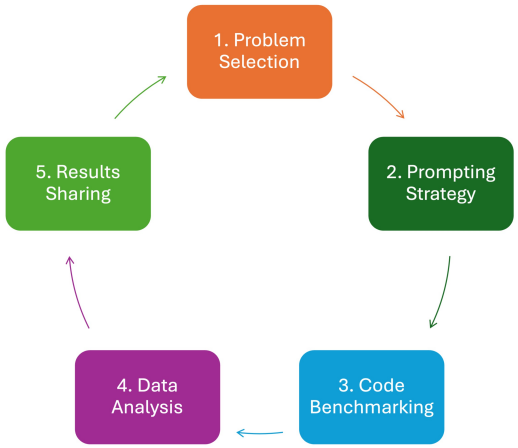

Figure 1. Experiment Approach for Comparing Stack Overflow and ChatGPT-4o Solutions

Figure 1 visualizes our experiment approach for comparing Stack Overflow and ChatGPT-4o solutions, which is described below:

1. **Problem Selection** –
   - Manual curation from Stack Overflow
   - Select Python questions (*e.g.*, array questions)
   - Avoid third-party Python libraries
2. **Prompting Strategy** –
   - Use question titles as prompts
   - Leverage ChatGPT-4o API for code generation
   - Generate up to 100 solutions per prompt
3. **Code Benchmarking** –
   - Measure runtime performance using Python's *timeit* package
   - Small (1,000), medium (10,000), and large (100,000) inputs
   - Generate 100 random inputs per size
4. **Data Analysis** –

[1] Our rationale for only using question titles as prompts for ChatGPT-4o both (1) reflects common real-world scenarios faced by developers and (2) assesses its ability to generate solutions based on limited information.

- Compare human and ChatGPT-4 solutions
- Assess efficiency, correctness, and scalability

5. **Results Sharing** –
   - Results repository: Github
   - Encourage reproduction and improvements

We measured the runtime performance of each code sample via Python's *timeit* package. Code samples were provided with small (1,000), medium (10,000), and large (100,000) inputs, which we progressively increased in size and measured the effects of scaling on generated code efficiency. For each input size, we generated 100 random inputs of the given size and tested the given code 100 times on each input using Python's *timeit* package.

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

3.1.2 Overview of the Coding Problems

Seven problems from Stack Overflow pertaining to array operations were selected for our analyses. These problems covered the following array-related challenges:

PA1-Find Missing Number – Identify missing number(s) in an unsorted array. This task involves determining which number(s) are absent from a sequence of integers within a given range.

PA2-Single Duplicate Finder – Detect a single duplicate number in an unsorted array. The challenge here is to find one number that appears more than once, without sorting the array first.

PA3-k Smallest Indices – Find the indices of the k smallest numbers in an unsorted array. This problem requires efficiently locating the positions of the smallest values in the array without sorting the entire array.

PA4-Sum Pairs – Count pairs of elements in an array with a given sum. The goal is to identify all unique pairs of numbers whose sum matches the target value, which is a common problem in algorithmic challenges.

PA5-Multiple Duplicate Finder – Find all duplicates in an array. This procedure involves scanning the array to identify and list all elements that occur multiple times, without sorting the array first.

PA6-Duplicate Remover – Remove all duplicates from an array. The solution eliminates any repeated values, leaving only distinct elements while maintaining the original order as much as possible.

PA7-Quicksort Implementation – Implement the Quicksort algorithm, which sort elements efficiently by selecting a pivot and recursively partitioning the array with respect to this pivot.

3.1.3 Prompting Strategies

In this experiment we applied the following prompting strategies to generate Python code via ChatGPT-4o:

1. **Naive approach**, which used only the title from Stack Overflow as the prompt, *e.g.*, "Generate a Python algorithm to find the indexes of the k smallest number in an unsorted array."
2. **Ask for speed approach**, which added a requirement for speed at the end of the prompt, *e.g.*, "Generate a Python algorithm to find the indexes of the k smallest number in an unsorted 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 array size grows, *e.g.*, "Generate a Python algorithm to find the indexes of the k smallest number in an unsorted array, where the implementation should be fast as the array size grows."
4. **Ask for the most optimal time complexity**, which prioritized achieving the most optimal time complexity, *e.g.*, "Generate a Python algorithm to find the indexes of the k smallest number in an unsorted array, where implementation should have the most optimal time complexity possible."
5. **Ask for the chain-of-thought [29]**, 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: Python algorithm to find the indexes of the k smallest number in an unsorted 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."
6. **Ask for the detailed chain-of-thought**, which guides ChatGPT-4o to follow a structured chain of thought, step-by-step, to achieve the best possible runtime performance, *e.g.*, "How can we approach the problem of Python algorithm to find the indexes of the k smallest number in an unsorted array with a time complexity O(1) runtime? We can follow the steps below in our chain of thought: (1) What is the problem statement? (2) What is the naive approach to Python algorithm to find the indexes of the k smallest number in an unsorted array? What is its time complexity? (3) Can we improve the time complexity to O(1)? If yes, how? (4) Can you provide an algorithm to Python algorithm to find the indexes of the k smallest number in an unsorted array? in O(1) time complexity? (5) Can you explain how the algorithm works step-by-step? (6) Are there any edge cases that must be considered for the algorithm to work correctly? (7) Can you provide an

example to demonstrate how the algorithm works? (8) How does the O(1) algorithm compare to other algorithms in terms of time? (9) Can you think of any potential limitations or drawbacks of the O(1) algorithm? (10) Then, describe how to implement the algorithm step-by-step in the fastest possible way in Python."

We prompted ChatGPT-4o 100 times with each prompt per coding problem, yielding up to 100 different coding solutions per prompt. In practice, fewer than 100 unique coding solutions were sometimes produced since ChatGPT-4o often generated logically equivalent programs. However, we tested the performance of all ChatGPT-4o-generated code and removed no duplicate solutions. If two different prompts had identical solutions, we benchmarked each and included 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

Figures 2, 3, and 4 depict our experiment results from evaluating the performance of code obtained from Stack Overflow and generated by ChatGPT-4o 100 times for all seven coding problems with three different input sizes: small (1,000), medium (10,000), and large (100,000). Figure 5 shows the average performance across all input sizes.

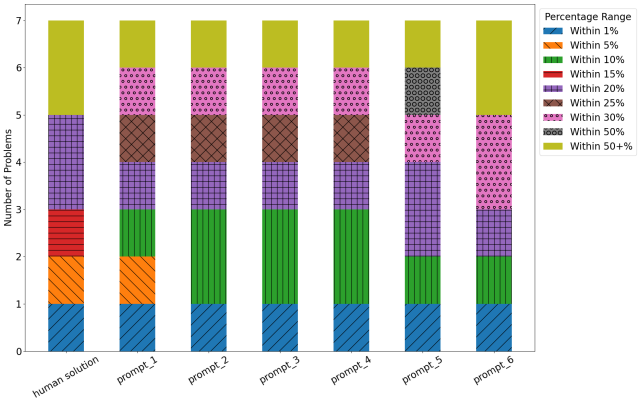

Figure 2. Number of Solutions within X% of the Best Runtime (Input Size 1,000)

These figures show the number of problems for each prompt where the best of the 100 solutions generated by each prompt was within 1%, 5%, 10%, etc. of the best solution found across all prompts and the human solution from Stack Overflow. Up to 601 solutions (6 prompts * 100 solutions per prompt + 1 human solution) were benchmarked for each problem. The "Best Runtime" solution shown in the figures was compared against the other solutions.

Figures 2, 3, 4, and 5 show how ChatGPT-4o selected the best-performing solution out of 100 attempts when employing detailed chain-of-thought reasoning in response to prompt #6. These solutions were competitive with—and often surpassed—the human solutions from Stack Overflow.

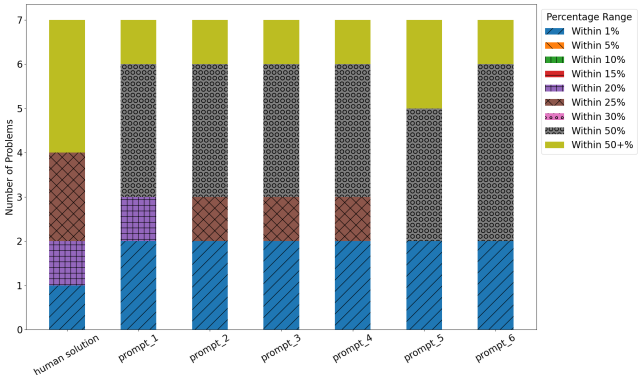

Figure 3. Number of Solutions within X% of the Best Runtime (Input Size 10,000)

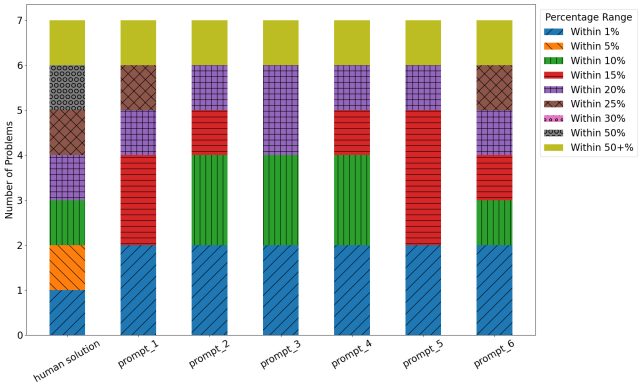

Figure 4. Number of Solutions within X% of the Best Runtime (Input Size 100,000)

This finding underscores the potential of LLMs in generating efficient solutions, especially when prompted using chain-of-thought reasoning.

The human solution was the best when we used the GPT-3.5 Turbo and GPT-4 models. However, the GPT-4o model outperformed the human solution for the problem *PA2-Single Duplicate Finder*, as shown in Figure 6. We used the title of the question as the input to ChatGPT-4o. All code samples produced code with respect to the title of the Stack Overflow post. However, since we directly translated the titles into prompts for ChatGPT-4o there may have been additional contextual information in the question that ChatGPT-4o used to further improve its solution, as discussed in Section 3.3.2.

Our results also reveal GPT-4o's significant performance improvement compared to its GPT-3.5 Turbo and GPT-4 predecessors, which underscores progress in AI-driven coding solutions. The human-crafted solution outperformed both GPT-3.5 Turbo and GPT-4 but was surpassed by GPT-4o in solving the *PA2-Single Duplicate Finder*. These results suggest recent LLMs are reaching a level of sophistication that can surpass human capabilities in certain coding tasks, particularly in terms of speed and efficiency.

Similarly, when evaluating the *PA1-Find Missing Number* problem, a distinct change in the hierarchy of solu-

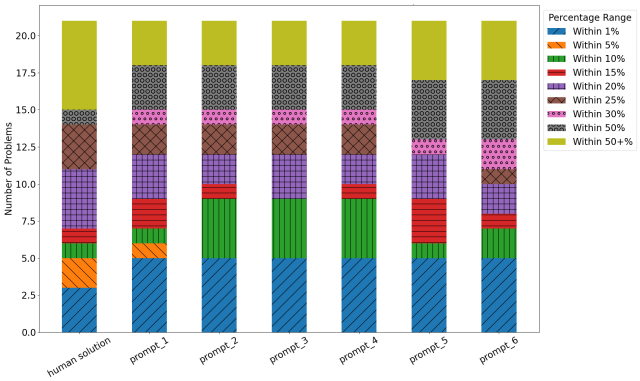

Figure 5. Number of Solutions within X% of the Best Runtime (All Input Sizes)

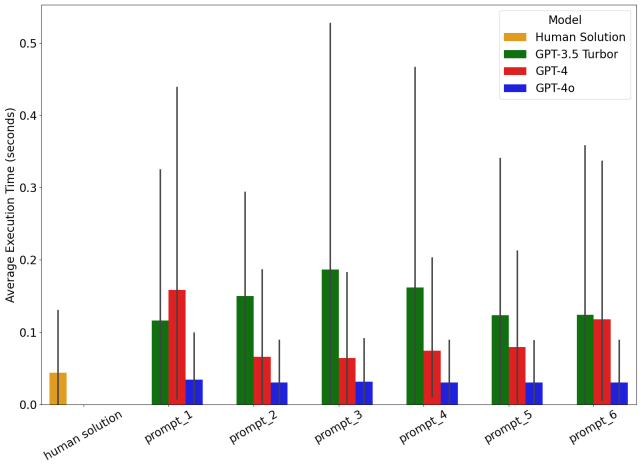

Figure 6. Comparison of Average Execution Time for Different Prompts in *PA2 Single Duplicate Finder*

tion efficiency was evident. As shown in Figure 7, the human solution was the *least* efficient in terms of execution time, which highlights scenarios where LLMs exceed human performance. GPT-4o outperformed all other models, including GPT-3.5 Turbo and GPT-4, showcasing its advanced LLM capabilities. Interestingly, when structured prompt engineering is applied—especially chain-of-thought reasoning—GPT-3.5 Turbo's capability to devise effective code solutions improves significantly.

In general, however, the most pronounced enhancement is seen with GPT-4o, which out-performs human solutions, GPT-4, and GPT-3.5 Turbo when applied using the same structured prompting techniques. This finding indicates recent advances in LLMs, especially in the realm of programming and problem-solving. The findings presented in Figure 7 confirm the superior performance of GPT-4o in optimizing code execution time and setting a new threshold in AI-assisted coding, which in turn will likely be surpassed with new releases of ChatGPT and other LLMs.

These results show GPT-4o consistently outperforms both humans and other AI models, which offer insights into

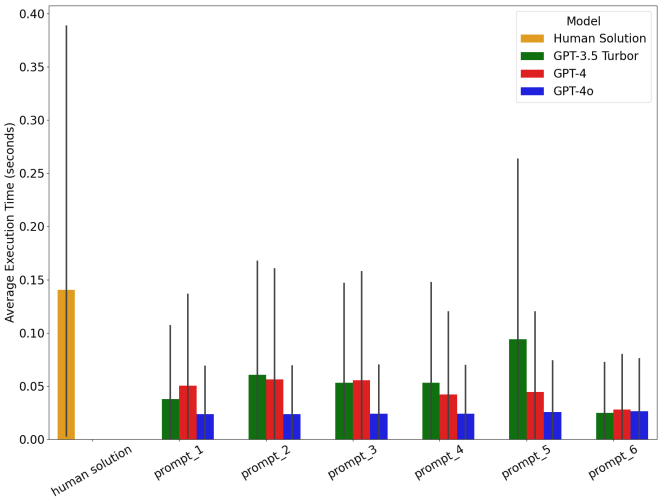

Figure 7. Comparison of Average Execution Time for Different Prompts in *PA1 Find Missing Number*

the evolution of LLMs for coding solutions. However, although LLMs like ChatGPT-4o can outpace human coders in certain instances, the creativity and specialized skill of human programmers remain invaluable in complex scenarios. This dynamic highlights the benefits of a synergistic approach, wherein human expertise is enhanced by the evolving capabilities of LLMs, thereby elevating the process of developing efficient coding solutions.

3.3 Threats to Validity

The threats to the validity of the experiment presented in this section 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 problems, each subjected to 100 test 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 benefit from benchmarks that associate

- Code needs (expressed as natural language requirements), questions, specifications, and rules with
- 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 on optimizing LLMs is maturing.

3.3.2 Prompt Construction

Prompt construction posed an additional threat to validity since our tests relied only on Stack Overflow question titles.

Incorporating no additional details from question bodies thus prevented ChatGPT-4o from using further context to inform its responses. Although we did not want ChatGPT-4o completing/improving fundamentally flawed code, this prompt design choice risked depriving ChatGPT-4o of information it could have used to generate better solutions.

3.3.3 Problem and Solution Scope

Another risk was the variety of coding problems we analyzed, which were relatively narrow in scope, data structures, and programming language. A wider range of problems are thus needed to ensure hidden risks regarding specific problem structures do not occur. In particular, we may not be aware yet of classes of problems that trigger LLMs to generate inefficient code. This risk is particularly problematic when generalizing our results to a wider range of algorithms and programming languages. In addition, focusing only on the ChatGPT family of LLMs limits our ability to apply our findings to other LLMs.

3.3.4 Selection Bias

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

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

Computer science and its application domains evolve continuously, motivating the need for more efficient and reliable automated systems capable of solving increasingly complex problems. To evaluate candidate solutions systematically, this section compares AutoGen version 0.3.1 and ChatGPT-4o, which are two generative AI-based tools that enable automated problem-solving. Our comparison evaluates the capability of ChatGPT-4o 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-4o is a recent addition to OpenAI's GPT family of LLMs and is adept at a 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

Both ChatGPT-4o and Autogen 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 tests are generated dynamically as part of the problem-solving process. Addressing this knowledge gap raises a critical question (stated as Q4 in Section 2): *How does AutoGen, with its systematic and structured LLM prompting, compare with the more flexible and generalized approach of ChatGPT-4o in terms of efficiency, accuracy, and adaptability in code generation?*

The study presented in this section aims to fill the current gap regarding the adaptability and precision of ChatGPT-4o and Autogen in dynamic testing environments. The absence of predefined tests means the evaluation of these AI tools 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 dynamically-created tests.

This section addresses these challenges to provide deeper insights into the capabilities of ChatGPT-4o and AutoGen. We also explore whether these AI tools can autonomously generate both problems and their corresponding tests, which is common practice in continuous integration pipelines and automated software development processes [1]. Our comparative analysis results evaluate the potential of these AI tools to enhance the field of automated software testing and development.

4.2 Dataset Overview and Analysis

Our dataset comprises a collection of 50 computer science problems, each characterized by a unique sequence number, a difficulty level (category), a problem type, and a detailed problem statement. These problems are classified into various categories that reflect different computer science fields, such as algorithm design, data structures, and computational theory. The problems are categorized by difficulty levels, ranging from easier to harder problems. The dataset and related information are available on our GitHub repository: `https://github.com/elnashara/GenAI-CodeEval`.

This dataset includes a broad spectrum of test cases for each problem, ensuring a comprehensive skill evaluation 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. As with the array study in Section 3, this method showcases dataset range, from fundamental algorithms like "Binary Search" to advanced techniques like "Depth-First Search."

The analysis of the distribution of computer science problems by type indicates the wide range of topics encompassed within the dataset. The pie chart shown in Figure 8

(adapted from [9]) depicts the percentage of problems in

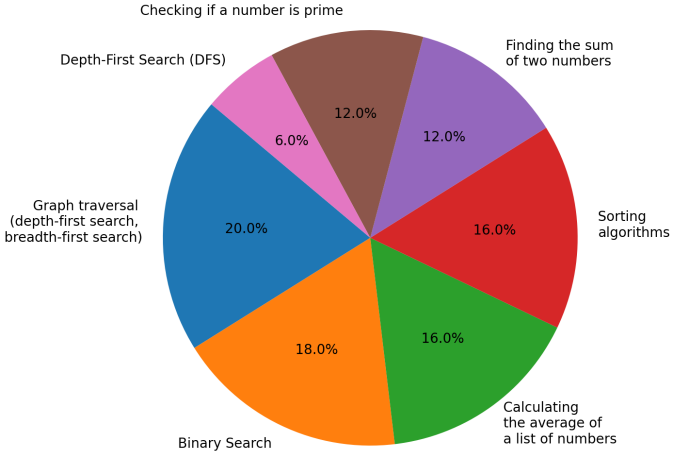

Figure 8. Distribution of Problem Types

each type, providing a visual representation of which areas are emphasized. This distribution is crucial for understanding the breadth and focus areas of our dataset.

The pie chart shown in Figure 9 (also presented in [9]) depicts 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 cat-

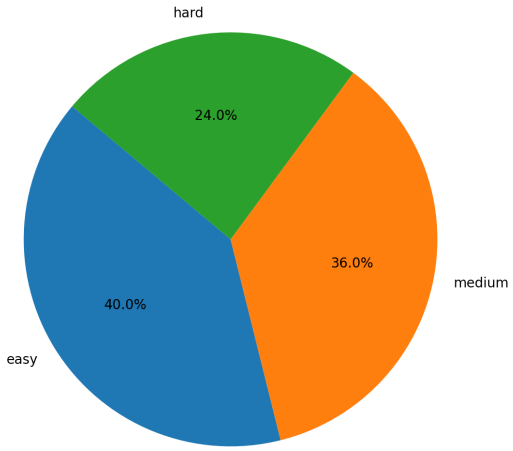

Figure 9. Distribution of Problems by Difficulty Level

egory, 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 covers the evolving landscape of automated problem-solving and algorithm generation, focusing on ChatGPT-4o and AutoGen. We employ a consistent structured prompting strategy to harness the capabilities of these AI tools by conveying problem requirements and context uniformly. In particular, this prompt facilitates direct comparison of ChatGPT-4o and Autogen in terms of their problem-solving efficiency, accuracy, and adaptability.

By employing this standardized prompt strategy across all tests, our study compares and contrasts the performance of these two AI tools in a controlled and comparable manner. Our problem-solving environment is dynamic, *i.e.*, tests are not static but generated in response to each unique problem. This study therefore evaluates the efficiency and accuracy of these AI tools under varying conditions.

4.3.1 Problem-Solving and Test Generation Approach

We apply prompt engineering [7] to guide ChatGPT-4o and AutoGen to interpret problem statements and generate corresponding solutions and tests. Figure 10 , as presented in [9], illustrates the structured prompt given to both AI tools, which enabled them to understand the given problem(s).

```
## 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 10. Structured Prompts for LLM-based Solution Generation in CS Problems.

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-4o and Autogen, focusing on their ability to generate solutions, as well as create relevant and comprehensive test cases.

4.3.2 Evaluating ChatGPT-4o and AutoGen

The evaluation of ChatGPT-4o and AutoGen involved the following two phases:

1. We assessed these AI tools' ability to interpret problem statements accurately and generate viable solutions.
2. We examined the appropriateness and thoroughness of the autonomously created test cases.

These test cases were vital to our evaluations since they represented the dynamic criteria that generated solutions were measured against. Our assessment compared the solutions and tests generated by each AI tool under identical problem conditions. This comparative analysis evaluated the adaptability, precision, and reliability of ChatGPT-4o and AutoGen in a dynamic testing environment where problems and their tests were both generated autonomously.

This study provided a nuanced understanding of ChatGPT-4o and AutoGen's capabilities in terms of automated problem-solving and test generation. Our work is particularly relevant in contexts like continuous integration pipelines and automated software development processes where the ability to autonomously generate and test solutions is vital. Our study findings provide insight into the potential role of AI tools in enhancing automated software testing and development.

4.4 Analysis of ChatGPT-4o Experiment Results

This experiment used ChatGPT-4o's solution generation capabilities to provide a systematic view of its performance across a range of computer science problems. To ensure a fair and accurate comparison, the tests used the same set of 50 distinct problems and identical prompts were used for both ChatGPT-4o and AutoGen. Figure 11 summarizes the effectiveness of the generated solutions. This figure shows

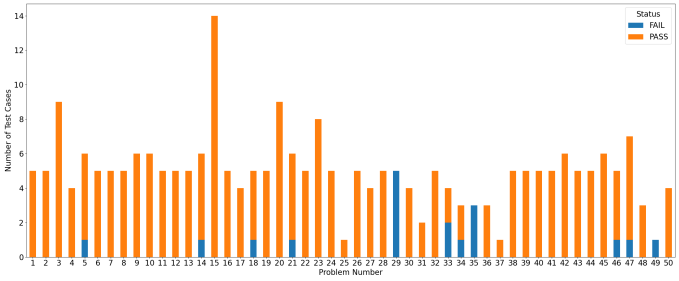

Figure 11. ChatGPT-4o: Pass Rate of Solutions

the number of pass and fail test cases for each of the 50 problems evaluated using ChatGPT-4o. Each bar represents a problem, with the orange sections indicating the number of test cases that passed and the blue sections representing the number of test cases that failed. Variation in the height of the bars varies indicates the different number of test cases run for each problem.

Figure 11 visualizes the success rate of ChatGPT-4o across a range of problem types. Most problems had more passes than failures. However, certain problems (*e.g.*, problem 29, 33, and 35) encountered a higher proportion of failures.

4.4.1 Overall Success Rate

ChatGPT-4o's overall success rate was 92.8%, as shown in Figure 12. This success rate indicates its ability to solve a

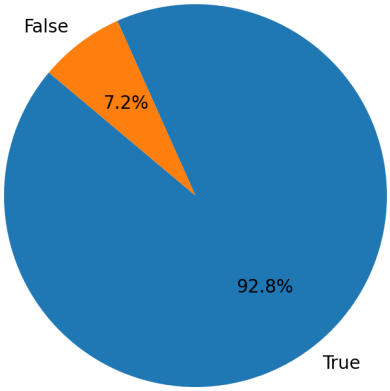

Figure 12. ChatGPT-4o: Pass Rate of Solutions

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

4.4.2 Error Analysis

Distinct patterns emerged when examining ChatGPT-4o'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 "The sum of weights must not be zero." and "item 'E' is not in list" These errors indicate that while ChatGPT-4o was proficient in many areas, improvements were needed in specific scenarios, particularly those involving input validation and handling exceptional cases.

4.4.3 Problem Difficulty vs. Success Rate

An interesting aspect of ChatGPT-4o's behavior is the correlation between problem difficulty and success rate. Surprisingly, 'medium' difficulty problems had a lower success rate (83.78%) compared to 'hard' (94.74%) and 'easy' (97.48%) difficulties, as shown in Figure 13. This finding suggests either (1) a potential discrepancy in the perceived versus actual complexity of the problems or (2) a higher adaptability of ChatGPT-4o in solving easy complexity tasks.

4.4.4 Problem Type Analysis

ChatGPT-4o's success rate also varied significantly across different problem types, as shown in Figure 14. Types such as *Binary Search, hard* and *Calculating the average of an array of numbers, easy* demonstrated a notably high success rate (over 94%), whereas *Graph traversal, medium* and *Sorting algorithms, medium* exhibited lower success rates. This variation highlighted ChatGPT-4o's strengths

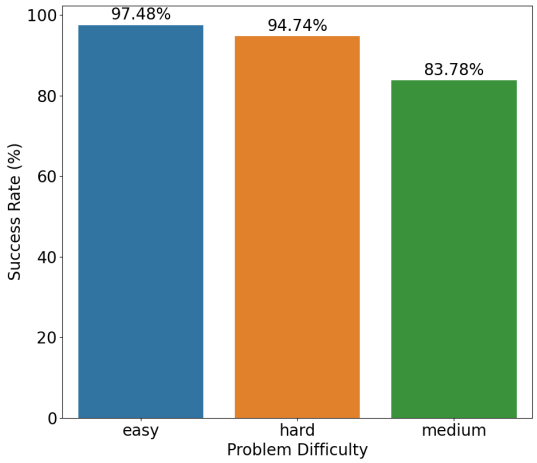

Figure 13. ChatGPT-4o - Problem Difficulty vs. Success Rate

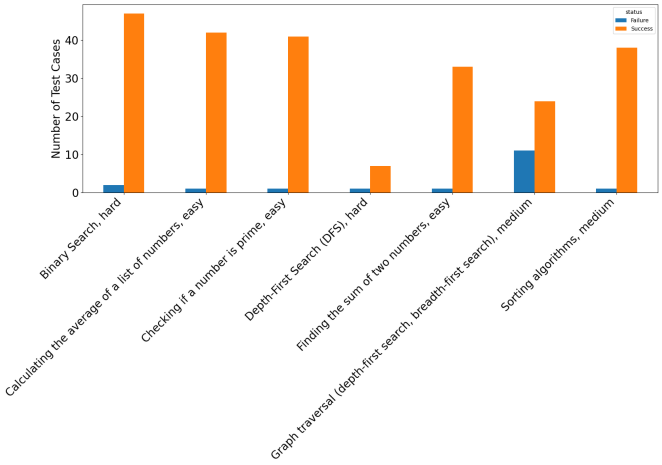

Figure 14. ChatGPT-4o - Success and Failure Analysis by Problem Type and Category

and weaknesses in different computational domains and provided opportunities for targeted improvements in specific areas of problem-solving.

4.4.5 Insights and Future Directions

Overall, ChatGPT-4o's experiment results revealed that it was highly effective in solving a wide range of computer science problems. However, insights gained from error analysis and variation in success/failure rates across problem types 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-4o. These findings help inform future efforts to optimize the solution generation capabilities of ChatGPT-4o.

4.5 Analysis of AutoGen Experiment Results

This experiment conducted on AutoGen 0.3.1 for computer science problems provided a wealth of data, allowing in-depth analysis of its performance. The dataset comprises results from tests conducted on 50 different computer science problems shown in Figure 15, where each test was evaluated across multiple parameters. Once again, the same

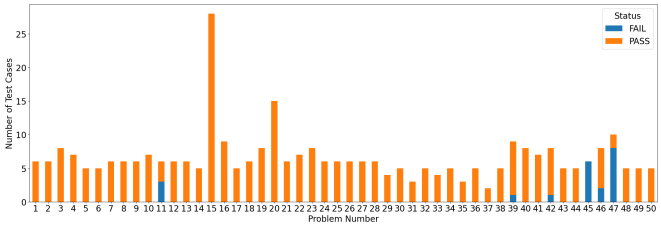

Figure 15. Number of Pass and Fail Test Cases for Each Problem Using AutoGen

set of tests was used for both ChatGPT-4o and Autogen to ensure a fair comparison of their performance.

4.5.1 Overall Success Rate

AutoGen achieved an overall success rate of 93.6%, as shown in Figure 16. This high percentage indicates that

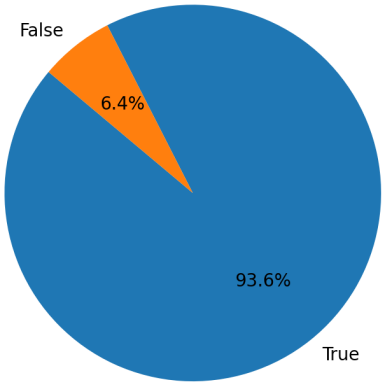

Figure 16. AutoGen: Pass Rate of Solutions

AutoGen solved the majority of the problems correctly by its auto-generated solutions. It thus 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. Figure 17 visualizes the success rates of solutions across different 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, which enables explicit comparison of success rates across different levels of problem complexity.

AutoGen's approach, characterized by structured LLM prompting, is highly effective for problems of varying complexity. We initially hypothesized that AutoGen would perform equally well in solving both 'easy' and 'hard' problems, leveraging its deep learning capabilities. We an-

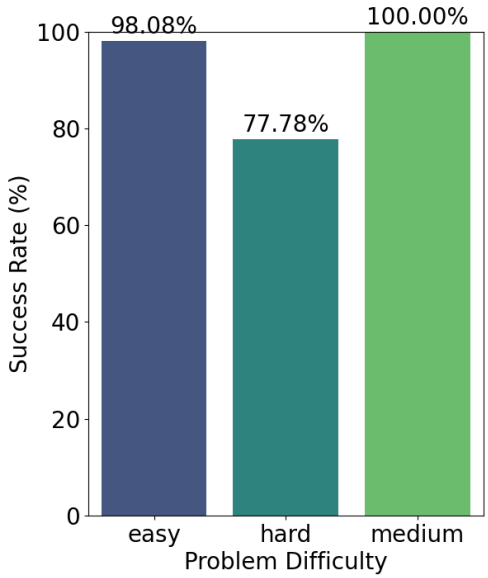

Figure 17. AutoGen - Problem Difficulty vs. Success Rate

ticipated a relatively higher success rate for 'hard' problems, based on the belief that the structured nature of LLM prompting would help mitigate their difficulty.

Contrary to our initial assumption, however, Figure 17 shows that the success rate for 'easy' problems was (98.08%), while 'hard' problems had a lower success rate of (77.78%). Surprisingly, 'medium' problems achieved a perfect success rate of (100.00%). These results suggest AutoGen is highly proficient in handling moderately complex problems but may incur challenges with harder tasks. Nonetheless, AutoGen still performs well across all problem categories, with relatively high success rates.

These findings offer insights into AutoGen's capabilities, showcasing its strength in managing 'medium' difficulty problems, with near-optimal performance on easy' problems. Although AutoGen's success rates for 'hard' problems are somewhat lower, it is generally effective, making it an effective tool for complex problem-solving scenarios.

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, as shown in Figure 18. One issue occurred with *binary search*, where AutoGen struggled to provide correct results consistently. This problem was primarily seen in the 'hard' difficulty category. The complexity of the algorithm and potential edge cases likely contributed to higher failure rates in this area.

Another challenge was observed in problems involving *calculating the average of a list of numbers*. AutoGen incurred errors in cases where it encountered 'NoneType' values in the dataset, leading to a mismatch in expected output types (*e.g.*, when a floating-point number was expected, but

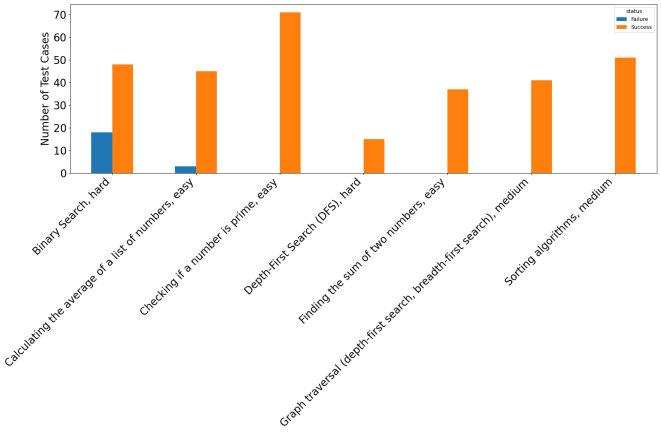

Figure 18. AutoGen - Success and Failure Analysis by Problem Type and Category

'NoneType' was found). This issue was prevalent in easier problems involving basic numerical operations.

Despite these specific challenges, AutoGen performed exceptionally well in other categories, achieving a perfect (100.00%) success rate in problems involving *sorting algorithms*, *depth-first search*, and simpler tasks like *checking if a number is prime* or *finding the sum of two numbers*. These results indicate that AutoGen is highly reliable in handling both simple and moderately complex problems, with difficulties mainly arising in highly specialized cases, such as binary search and specific numeric operations.

These results suggest that while AutoGen excels in most problem categories, future improvements should focus on enhancing its performance in handling binary search and certain numerical data processing edge cases. Addressing these issues will improve AutoGen's robustness and accuracy in generating solutions for broad range of problems.

4.6 Comparative Analysis of ChatGPT-4o and AutoGen Experiment Results

Conducting a detailed comparative analysis between the ChatGPT-4o and AutoGen experiment results revealed several key distinctions and similarities, as well as offered insightful perspectives on the performance and application of each system. Our analysis begins by examining the overall success rates of both AI tools shown in Figure 19.

Figure 19 shows that AutoGen achieved a slightly higher success rate (93.6%) compared to ChatGPT-4o (92.8%). Although the difference is minimal, it indicates that AutoGen outperforms GPT-4o in terms of its overall ability to generate correct solutions. This result suggests that while both AI tools are reliable, AutoGen has a slight edge in handling the problem set in this experiment.

Figure 20 shows that differences with error analysis become more pronounced between AutoGen and GPT-4o. AutoGen recorded (21 out of 329) failed test cases, while GPT-4o encountered (18 out of 250) failed cases. This finding indicates that the total number of test cases generated by

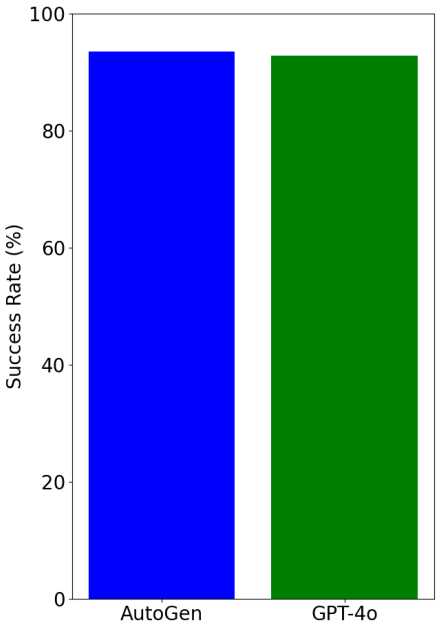

Figure 19. Success Rate Comparison

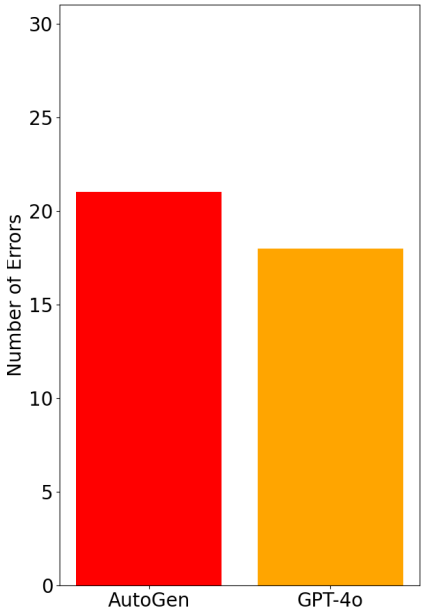

Figure 20. Number of Errors Comparison

Metric	ChatGPT-4o	AutoGen
Total Success Rate	Slightly lower (~92.8%)	Higher (~93.6%)
Hard Problem Success Rate	Higher (~94.74%)	Lower (~77.78%)
Error Analysis	Fewer errors, with specific exceptions reported	More errors, mostly without detailed exception information
Complexity Handling	Advanced error handling and input validation	Focuses on basic arithmetic with some challenges in complex scenarios
Ideal Use Case	Advanced learning, comprehensive testing environments	Educational use, basic programming challenges

Figure 21. Comparative Analysis of ChatGPT-4o and AutoGen

AutoGen was higher than those generated by GPT-4o, despite both AI tools using the same prompts. This result also implies that AutoGen generated more complex test cases, which in turn resulted in a higher number of errors.

Conversely, AutoGen's errors were less detailed, with most errors not providing specific exception information. In contrast, GPT-4o provided more informative error messages, including specific exceptions like input validation errors. This added transparency in error handling gives GPT-4o an advantage with respect to debugging and understanding the root causes of failures.

We also analyzed the complexity of problems and the handling of solutions by both AI tools. Although tasked with similar problems, ChatGPT-4o's solutions demonstrated more advanced capabilities in error handling and input validation, particularly in scenarios involving complex logic or edge cases. Conversely, AutoGen's strength lies in basic programming tasks and educational use cases, where it performed exceptionally well with a perfect (100.0%) success rate.

Overall, AutoGen exhibited a higher total success rate (93.6%) and proved quite effective for basic programming challenges, as shown in Figure 21. However, it recorded more errors (21 failed cases) compared to GPT-4o (18 failed cases). This difference suggests that while AutoGen is suitable for educational purposes, it may need improvements in error handling and detailed exception reporting.

In contrast, GPT-4o excelled in handling complexity, particularly in advanced error handling and input validation, making it a better candidate for comprehensive testing environments and more advanced learning scenarios. These differences highlight the specific strengths of each AI system and their potential use cases, depending on the context and complexity of the given programming tasks.

5 Related Work

This section compares our research with related work. The Venn diagram in Figure 22 categorizes existing related work in LLM research across three main dimensions: *Prompt Engineering*, *Security and Reliability*, and *Impact on Development Workflows*. Each circle represents one of these research areas, with the overlaps highlighting studies that integrate two or more areas. The following breakdown details the overlaps between the three research areas, providing examples of studies that fall within each intersection:

- The overlap between *Prompt Engineering* and *Security and Reliability* includes research on AI-driven code reliability, *e.g.*, how well LLMs can generate secure code based on prompts.
- The intersection of *Security and Reliability* and *Impact on Development Workflows* features studies that focus on secure code deployment practices and their effect on developer workflows.

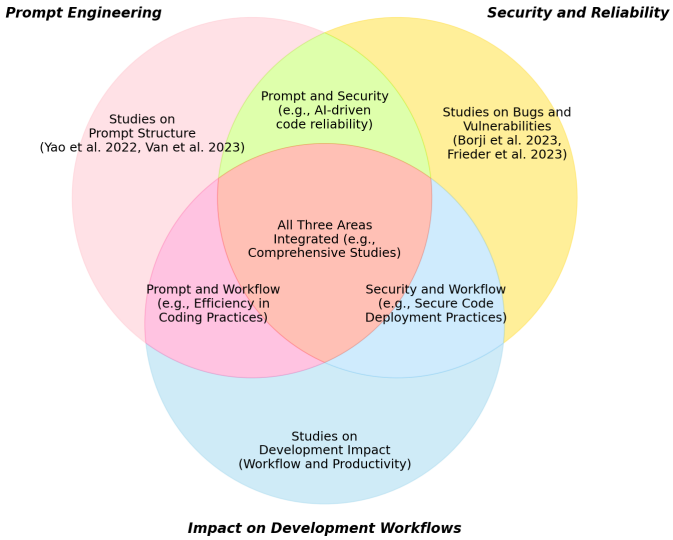

Figure 22. Overview of Related Work in LLM Research.

- Similarly, the overlap between *Prompt Engineering* and *Impact on Development Workflows* shows how prompts can improve coding efficiency.
- The center area where all three circles intersect represents comprehensive studies that consider all three aspects of prompt quality, security concerns, and workflow efficiency.

The Venn diagram in Figure 22 also visualizes the interconnectedness of related work. Moreover, this diagram shows how our research not only builds on each individual dimension but also contributes to the growing body of work that integrates all three aspects to provide a more holistic understanding of LLM impacts on code development.

5.1 Advances in Prompt Engineering

The evolution of LLMs in code generation has been pivotal, particularly in the discipline of prompt engineering. This discipline focuses on crafting effective natural language inputs for LLMs, enabling them to solve complex problems across diverse domains [7]. Recent studies emphasize the importance of prompt structure and leverage external tools and methods to enhance the capabilities of LLMs in coding tasks.

For example, Yao et al. (2022) [28] showed how integrating LLMs with coding frameworks can improve their utility. Likewise, Van et al. (2023) [24] maximized the inherent capabilities of LLMs to generate more complex and efficient code structures. These prompt engineering advances provide promising results in domains like mathematics, where straightforward prompting often falls short, necessitating more sophisticated approaches for better outcomes [12].

Our work explores the impact of refined prompt design on LLM-generated code performance, building on related work that primarily applies direct queries, such as those found on Stack Overflow. Our approach serves as a baseline for comparison, but we focus on investigating how refined techniques can yield more efficient and accurate solutions. The field of prompt engineering offers immense potential for improving LLM performance and our work contributes to this area by showcasing the benefits of structured and nuanced prompting strategies.

5.2 Security and Reliability of LLM-Generated Code

The reliability and security of LLM-generated code have emerged as critical areas of focus. Recent studies, including those by Borji et al. (2023) [5] and Frieder et al. (2023) [12], identified various bugs and vulnerabilities inherent in AI-generated code. Comparative analyses, such as those by Asare et al. (2022) [2], explored the security profiles of human-written versus AI-generated code, revealing significant insights into the strengths and weaknesses of both approaches.

Jalil et al. (2023) [15] focused on how LLMs like ChatGPT can be applied to software testing education, discussing both the promises and potential pitfalls related to the security of generated code. They emphasized that while LLMs can automate testing tasks, they can also introduce vulnerabilities if not monitored carefully. Similarly, Nair et al. (2023) [18] examined the process of generating secure hardware using LLMs and highlighted specific challenges, such as common weaknesses and exposures (CWEs), that can arise when LLMs are tasked with producing code for security-sensitive applications.

The findings in this related work are essential for understanding the trade-offs involved in using LLMs for software development, particularly in contexts where code security and reliability are paramount.

5.3 Impact on Software Development Workflows and Productivity

The impact of LLMs on software development workflows and productivity is an emerging area of interest. Research is examining how LLMs influence the software development lifecycle—from design to deployment—and their potential to accelerate development processes while maintaining or enhancing code quality. For instance, Chen et al. (2021) [11] evaluated the effectiveness of LLMs in automating code generation, showing how developers can leverage AI tools to handle repetitive tasks, thereby freeing up time for more creative problem-solving. Their findings aligned with Nair et al. (2023) [18], who showed how LLMs can integrate seamlessly into development pipelines to streamline testing and debugging processes, further improving workflow efficiency.

Peng et al. (2023) [21] examined the use of AI tools (such as GitHub Copilot [14]). They highlighted how these tools can boost developer productivity by suggesting code

snippets in real-time, which accelerates development without sacrificing code quality. These AI tools also helped reduce cognitive load on developers by handling tedious tasks, allowing them to focus on higher-level system design and architecture concerns.

5.4 Studies on LLM Applications in Software Development

Recent studies are integrating multiple research areas to provide a holistic view of LLM applications in software development. For example, Pearce et al. (2022) [20] explored security and ethical implications of LLM-generated code, focusing on the challenges posed by AI-driven systems in maintaining code security and reducing vulnerabilities. Their study highlighted the need for more robust validation and verification techniques when using LLMs in critical software applications to ensure reliability is maintained throughout the software development lifecycle.

Similarly, Bommasani et al. (2021) [4] provided a broad analysis of foundation models like GPT-3, examining their capabilities and limitations across various domains, including software development. Their work underscored the potential for LLMs to enhance productivity and efficiency in coding tasks. However, they also cautioned about the risks of relying too heavily on AI for code generation, especially in complex, safety-critical systems.

These studies emphasized the need for a balanced approach to LLM integration, acknowledging both opportunities and challenges. As LLMs continue to evolve, research is pushing the boundaries of AI-driven code generation, with further studies required to investigate their long-term impact on software development practices and the intersection of AI and software engineering. For example, Terragni et al. (2024) [23] explored how LLMs are transforming various aspects of software engineering, including requirements gathering, code generation, and software design. They viewed LLMs as tools that can assist developers by automating tasks, offering code suggestions, and helping with error handling, thereby improving productivity and efficiency throughout the software lifecycle.

6 Concluding Remarks

In this paper we analyzed programming automation techniques and tools by evaluating top Stack Overflow solutions against those generated by ChatGPT-4o and comparing the capabilities of ChatGPT-4o with AutoGen. This section summarizes lessons learned from our work thus far and outlines future work on this topic.

Key lessons learned from our research include:

- **ChatGPT-4o produced solutions competitive to humans**. We observed that ChatGPT-4o could produce solutions competitive with—and sometimes superior to—human-crafted ones. Our holistic approach enhanced ChatGPT-4o's existing problem-solving and code generation capabilities. While AutoGen achieved a slightly higher overall success rate (93.6% vs. 92.8%), ChatGPT-4o exhibited greater versatility, especially in more complex tasks involving error handling and detailed input validation, which makes it an ideal candidate for environments requiring advanced programming solutions.
- **Effective prompting was crucial for solving complex problems**. Our analysis showed that ChatGPT-4o generated solutions competitive with—or superior to—top Stack Overflow answers when prompted effectively, especially when prompts apply chain-of-thought reasoning. This finding underscored the potential of LLMs in complex coding tasks, but also revealed limitations when minimal context, such as Stack Overflow titles, was used in isolation. Effective prompt patterns [26] and prompt engineering [25] are thus essential to fully leverage LLM code generation capabilities. Our results in Section 3 also showed that prompting multiple times and selecting the best solution was a promising means for software developers to optimize performance-critical code sections via LLMs.
- **Choosing between AI tools depends on application needs**. Application-specific requirements should guide the choice between ChatGPT-4o and AutoGen. ChatGPT-4o's robust (92.8%) success rate showcased its ability to handle complex programming tasks, such as advanced error handling and input validation, making it suitable for testing environments or scenarios involving intricate logic and edge cases. However, its error diagnosis and reporting need further refinement. In contrast, AutoGen recorded a slightly higher overall success rate (93.6%) and excelled in basic programming tasks and educational use cases. However, its error messages lacked detailed exception reporting, which limited its transparency in debugging. ChatGPT-4o may thus be more suitable for applications needing robust error handling and debugging capabilities, while AutoGen may be better suited for educational scenarios or basic programming tasks that prioritize simplicity and high success rates.
- **Advanced data analysis enables broader solution exploration**. A key attribute of ChatGPT-4o-based code generation was its ability to search many coding solutions. Developers will likely use LLM-based tools like Advanced Data Analysis (ADA) [19] and Auto-GPT [27] to generate and analyze multiple solutions per query, as discussed in Section 4. Future AI tools should therefore enable defining metrics and automatically prompting until a quality threshold is met, a prompt limit is reached, and/or time runs out.

Our future work will explore 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 for CS education and professional software development.

Acknowledgements

We used ChatGPT-4o's Advanced Data Analysis (ADA) capability to generate code for the data visualizations and filter the data sets.

References

[1] S.A.I.B.S. Arachchi and Indika Perera. Continuous Integration and Continuous Delivery Pipeline Automation for Agile Software Project Management. In *2018 Moratuwa Engineering Research Conference (MERCon)*, pages 156–161, 2018.

[2] Owura Asare, Meiyappan Nagappan, and N Asokan. Is Github's Copilot as Bad as Humans at Introducing Vulnerabilities in Code? *arXiv preprint arXiv:2204.04741*, 2022.

[3] Yejin Bang, Samuel Cahyawijaya, Nayeon Lee, Wenliang Dai, Dan Su, Bryan Wilie, Holy Lovenia, Ziwei Ji, Tiezheng Yu, Willy Chung, et al. A Multitask, Multilingual, Multimodal Evaluation of ChatGPT on Reasoning, Hallucination, and Interactivity. *arXiv preprint arXiv:2302.04023*, 2023.

[4] Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. On the Ppportunities and Risks of Foundation Models. *arXiv preprint arXiv:2108.07258*, 2021.

[5] Ali Borji. A Categorical Archive of ChatGPT Failures. *arXiv preprint arXiv:2302.03494*, 2023.

[6] Anita Carleton, Mark H. Klein, John E. Robert, Erin Harper, Robert K Cunningham, Dionisio de Niz, John T. Foreman, John B. Goodenough, James D. Herbsleb, Ipek Ozkaya, and Douglas C. Schmidt. Architecting the Future of Software Engineering. *IEEE Computer*, 55(9):89–93, 2022.

[7] Banghao Chen, Zhaofeng Zhang, Nicolas Langrené, and Shengxin Zhu. Unleashing the Potential of Prompt Engineering in Large Language Models: A Comprehensive Review, 2023.

[8] Gabriele De Vito, Stefano Lambiase, Fabio Palomba, Filomena Ferrucci, et al. Meet C4SE: Your New Collaborator for Software Engineering Tasks. In *2023 49th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 235–238, 2023.

[9] Ashraf Elnashar, Max Moundas, Douglas Schmidt, Jesse Spencer-Smith, and Jules White. Evaluating the performance of llm-generated code for chatgpt-4 and autogen along with top-rated human solutions. In *Proceedings of the 19th International Conference on Software Technologies - ICSOFT*, pages 258–270. INSTICC, SciTePress, 2024.

[10] Jessica López Espejel, El Hassane Ettifouri, Mahaman Sanoussi Yahaya Alassan, El Mehdi Chouham, and Walid Dahhane. 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, 2023.

[11] Mark Chen et al. Evaluating Large Language Models Trained on Code. *arXiv: 2107.03374*, 2021.

[12] Simon Frieder, Luca Pinchetti, Ryan-Rhys Griffiths, Tommaso Salvatori, Thomas Lukasiewicz, Philipp Christian Petersen, Alexis Chevalier, and Julius Berner. Mathematical Capabilities of ChatGPT. *arXiv preprint arXiv:2301.13867*, 2023.

[13] Louie Giray. Prompt Engineering with ChatGPT: A Guide for Academic Writers. *Annals of biomedical engineering*, 51(12):2629—2633, December 2023.

[14] GitHub. GitHub CoPilot. https://github.com/features/copilot, 2024.

[15] Sajed Jalil, Suzzana Rafi, Thomas D LaToza, Kevin Moran, and Wing Lam. ChatGPT and Software Testing Education: Promises & Perils. *arXiv preprint arXiv:2302.03287*, 2023.

[16] Jarosław Krochmalski. *IntelliJ IDEA Essentials*. Packt Publishing Ltd, 2014.

[17] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing. *ACM Computing Surveys*, 55(9):1–35, 2023.

[18] Madhav Nair, Rajat Sadhukhan, and Debdeep Mukhopadhyay. Generating Secure Hardware using ChatGPT Resistant to CWEs. *Cryptology ePrint Archive*, 2023.

[19] Mohamed Nejjar, Luca Zacharias, Fabian Stiehle, and Ingo Weber. Llms for science: Usage for code generation and data analysis, 2024.

[20] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 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, 2022.

[21] Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. The impact of ai on developer productivity: Evidence from github copilot, 2023.

[22] Sebastian Porsdam Mann, Brian D Earp, Nikolaj Møller, Suren Vynn, and Julian Savulescu. AUTOGEN: A Personalized Large Language Model for Academic Enhancement—Ethics and Proof of Principle. *The American Journal of Bioethics*, 23(10):28–41, 2023.

[23] Valerio Terragni, Partha Roop, and Kelly Blincoe. The Future of Software Engineering in an AI-Driven World. *arXiv 2406.07737*, 2024.

[24] Eva AM van Dis, Johan Bollen, Willem Zuidema, Robert van Rooij, and Claudi L Bockting. ChatGPT: Five Priorities for Research. *Nature*, 614(7947):224–226, 2023.

[25] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C. Schmidt. A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT. In *Proceedings of the 30th Pattern Languages of Programming (PLoP) conference*, Allerton Park, IL, October 2023.

[26] Jules White, Sam Hays, Quchen Fu, Jesse Spencer-Smith, and Douglas C Schmidt. ChatGPT Prompt Patterns for Improving Code Quality, Refactoring, Requirements Elicitation, and Software Design. In *Generative AI for Effective Software Development*, pages 71–108. Springer, 2024.

[27] Hui Yang, Sifu Yue, and Yunzhong He. Auto-gpt for online decision making: Benchmarks and additional opinions, 2023.

[28] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing Reasoning and Acting in Language Models. *arXiv preprint arXiv:2210.03629*, 2022.

[29] Zhuosheng Zhang, Aston Zhang, Mu Li, and Alex Smola. Automatic Chain of Thought Prompting in Large Language Models. *arXiv arXiv:2210.03493*, 2022.