



Baher El-Banna, Bsc

Investigating on the Use of ChatGPT in Software Engineering

Master's Thesis

to achieve the university degree of
Master of Science

Master's degree programme: Software Engineering and Management

submitted to

Graz University of Technology

Supervisor

Univ.-Prof. Dipl.-Ing. Dr.techn Franz Wotawa

Institute of Software Engineering and Artificial Intelligence

Head: Univ.-Prof. Dipl.-Ing. Dr.techn Franz Wotawa

Graz, January 2025

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my supervisor, Univ.-Prof. Dipl.-Ing. Dr.techn Franz Wotawa, for his continuous guidance, support, and valuable feedback throughout this thesis. His expertise, constructive criticism, and insightful suggestions have greatly influenced the progress and focus of my research. I deeply appreciate the time and effort he dedicated to helping me grow.

I am immensely thankful to my family and my wife, for their unwavering encouragement, patience, and belief in my abilities. Their support has been a constant source of motivation. A special thanks to my wife for inspiring me to persist, reviewing my thesis, and providing thoughtful feedback that greatly improved its clarity and quality.

Thank you all for being part of this important chapter of my life.

Abstract

In the ever-evolving field of software engineering, artificial intelligence (AI) technologies have led to profound transformative advancements, reshaping traditional approaches to development, debugging, and problem-solving. This thesis analyzes the capabilities of ChatGPT, a state-of-the-art large language model (LLM), in software engineering.

The study begins with an introduction that describes the content of the thesis, including the research objectives and experimental methods. Subsequently, the study reviews relevant literature on ChatGPT in software engineering to contextualize its focus. An overview of ChatGPT follows, covering its fundamental principles, potential use cases, benefits, and limitations.

Central to this research is an empirical evaluation of ChatGPT's proficiency across various software engineering tasks of different complexities. The tasks, solved in multiple programming languages, include challenges from basic algorithmic problems to complex software development projects, ranging from game development to cyber-security. Through a series of systematic experiments, categorized by difficulty, the study assesses ChatGPT's performance and capabilities to provide innovative solutions, highlighting its strengths and weaknesses.

The thesis comprehensively analyzes the research findings, supported by detailed documentation and examples. The findings demonstrate ChatGPT's potential to address diverse challenges encountered in software engineering. They emphasize the model's role in enhancing developer productivity, enabling collaborative coding, and driving advancements in AI-assisted software development. Additionally, the study identifies the system boundaries and potential areas of improvement associated with integrating ChatGPT in software engineering.

In conclusion, this thesis contributes to the evolving field of AI-assisted software engineering. They offer valuable insights into ChatGPT as an auxiliary tool in software development processes. The research opens new paths for AI-assisted software development, fostering innovation and addressing problems.

Kurzfassung

Im dynamischen Bereich der Softwareentwicklung haben fortschrittliche Technologien wie künstliche Intelligenz (KI) zu bedeutenden Veränderungen geführt, die traditionelle Ansätze für Entwicklung, Debugging und Problemlösung neu definieren. Diese wissenschaftliche Arbeit untersucht die Fähigkeiten von ChatGPT, einem Large Language Model (LLM), im Kontext der Softwareentwicklung.

Die Studie beginnt mit einer Einleitung, die einen Überblick über den Aufbau der Arbeit, die Forschungsziele und das experimentelle Setup sowie Systemanforderungen bietet. Anschließend wird der Stand der Forschung zu ChatGPT in der Softwareentwicklung analysiert, um den Kontext und die Relevanz der Untersuchung zu verdeutlichen. Es folgt eine Darstellung von ChatGPT, einschließlich seiner Grundfunktionalitäten, potenzieller Anwendungsbereiche, Vorteile und Grenzen.

Softwareentwicklung. Die untersuchten Aufgaben, die in verschiedenen Programmiersprachen gelöst werden, reichen von grundlegenden algorithmischen Problemen bis hin zu komplexen Softwareprojekten, wie etwa Spieleentwicklung und Cybersicherheit. Durch eine Reihe systematischer Experimente, die nach Schwierigkeitsgrad kategorisiert sind, bewertet die Studie die Fähigkeit von ChatGPT, innovative Lösungen bereitzustellen, und beleuchtet dessen Stärken und Schwächen.

Im Hauptteil der Arbeit werden die Forschungsergebnisse umfassend analysiert und durch eine detaillierte Dokumentation ergänzt. Die Ergebnisse zeigen das Potenzial von ChatGPT auf, komplexe Herausforderungen in der Softwareentwicklung zu bewältigen. Besonders hervorgehoben wird die Rolle des Modells bei der Steigerung der Produktivität von Entwicklern, der Förderung kollaborativer Programmierung und der Weiterentwicklung KI-gestützter Softwareentwicklung. Zudem werden die Systemgrenzen und mögliche Verbesserungsbereiche bei der Integration von ChatGPT identifiziert.

Abschließend leistet diese empirische Studie einen Beitrag zur Weiterentwicklung der KI-gestützten Softwareentwicklung und bietet wertvolle Einblicke in die Fähigkeiten und Einschränkungen von ChatGPT als Hilfswerkzeug in der Softwareentwicklung. Die Forschung eröffnet neue Perspektiven für zukünftige Innovationen und markiert einen Meilenstein in der KI-unterstützten Softwareentwicklung und Problemlösung.

Contents

1	Introduction	1
1.1	Research Objectives and Experimental Methods	2
1.2	Requirements	4
2	Related Research	5
2.1	Prompt Engineering with ChatGPT	6
2.2	Software Engineering with ChatGPT	8
2.3	Debugging with ChatGPT	10
2.4	Ethical aspects when using ChatGPT	12
2.5	Conclusion	14
3	ChatGPT	15
3.1	Use-Cases and Benefits	16
3.2	Issues and Boundaries	18
3.3	Summary	20
4	Experiments	22
4.1	Difficulty - Easy	24
4.1.1	Two Sums	24
4.1.2	Add Binary	24
4.1.3	Concatenation of Array	24
4.1.4	Removed Linked List Element	25
4.1.5	Implement Stack with Queue	25
4.2	Difficulty - Medium	26
4.2.1	Convert an Array Into a 2D Array	26
4.2.2	Total Distance Traveled	26
4.2.3	Validate IP Addresses	27
4.2.4	Array With Stack Operations	27
4.3	Difficulty - Hard	28
4.3.1	Encrypt and Decrypt Strings	28
4.3.2	Regex Matching	28
4.3.3	Sudoku Solver	29
4.4	Difficulty - Very Hard	30
4.4.1	AI creates Assistance AI	30
4.4.2	EvilTwin Wi-Fi	30
4.4.3	Arcade Games App	31

5	Experimental Procedure	32
5.1	Task Descriptions	37
5.2	Building Prompts	39
5.2.1	Mandatory Elements – Task and Context	41
5.2.2	Important Elements - Persona and Examples	41
5.2.3	Optional Elements - Format and Tone	41
5.3	Providing Additional Information	42
5.4	Testing Methodologies and Error Handling	45
5.5	Iterative Task Solving	46
5.6	Handling System Boundaries	46
5.7	Task Evaluation Process and Documentation	47
6	Results of the Experiment	48
6.1	Results (Easy)	49
6.2	Results (Medium)	52
6.3	Results (Hard)	55
6.4	Results (Very Hard)	57
6.5	Distribution of Prompt Variations Across Tasks	59
6.6	Summary	60
7	Discussion	63
7.1	Potential Relevance of ChatGPT in Software Engineering	64
7.1.1	Continuous Learning and Adaptation	64
7.1.2	Multi-Language Proficiency	67
7.1.3	Collaborative Coding	71
7.1.4	Natural Language Processing	72
7.1.5	Code Analysis	78
7.1.6	Code Optimization	83
7.1.7	Documentation	87
7.1.8	Potential for innovation	89
7.2	Limitations of ChatGPT in Software Engineering	90
7.2.1	Dependency on Input Clarity	91
7.2.2	Difficulty in solving complex problems	92
7.2.3	Inconsistent Performance Across Programming Languages	93
7.2.4	Error Handling	96
7.2.5	Non-deterministic Output	97
7.2.6	Potential to bypass restrictions	98
7.2.7	Hallucination	100
8	Conclusion and Future Outlook	102
	Bibliography	103

List of Figures

4.1	Sudoku Example [28]	29
4.2	Sudoku Example Solved [28]	29
5.1	Code Generation Workflow with ChatGPT	33
5.2	Documentation Workflow with ChatGPT	35
7.1	Comparison Python vs Erlang: Number of tasks solved	94
7.2	Comparison Python vs Erlang: Performance	95

List of Tables

4.1	List of tasks separated by difficulty	23
6.1	Solved Tasks (Easy)	49
6.2	Tests Passed (Easy)	49
6.3	Time Tracking (Easy)	50
6.4	Runtime Performance (Easy)	50
6.5	Memory Efficiency (Easy)	51
6.6	Solved Tasks (Medium)	52
6.7	Tests Passed (Medium)	52
6.8	Time Tracking (Medium)	53
6.9	Runtime Performance (Medium)	53
6.10	Memory Efficiency (Medium)	54
6.11	Solved Tasks (Hard)	55
6.12	Tests Passed (Hard)	55
6.13	Time Tracking (Hard)	56
6.14	Runtime Performance (Hard)	56
6.15	Memory Efficiency (Hard)	57
6.16	Solved Tasks (Very Hard)	58
6.17	Tests Passed (Very Hard)	58
6.18	Time Tracking (Very Hard)	58
6.19	Prompt Variations Used by Difficulty	60

Chapter 1

Introduction

Integrating artificial intelligence (AI) technologies has driven significant advancements across various fields, including software engineering. Among these innovations, ChatGPT, developed by OpenAI, has emerged as a groundbreaking tool with the potential to transform software development practices. Its ability to understand and generate human-like text has garnered substantial attention from researchers and developers, who are exploring its applications in enhancing productivity, automating tasks, and improving problem-solving processes in software engineering. [49][1][36]

As software engineering evolves to meet the growing demand for efficient and scalable solutions, tools like ChatGPT have demonstrated the ability to streamline workflows, assist in debugging, and enhance developer collaboration. This thesis investigates the utilization of ChatGPT, an advanced large language model (LLM), to address real-world programming challenges, including code generation, documentation, and debugging. Furthermore, it evaluates existing studies on ChatGPT’s integration into software engineering, highlighting its foundational principles, practical applications, and ethical implications, such as mitigating biases, ensuring reliability, and maintaining human oversight in AI-driven workflows.

The central research question guiding this thesis is:

Can ChatGPT function autonomously to handle diverse programming tasks, or is it more effective as a collaborative tool in software development workflows, considering its strengths, limitations (e.g. hallucinations), and adaptability across task complexity, programming languages, and real-world scenarios?

This question is explored through an empirical analysis of ChatGPT’s performance across various software development tasks of increasing complexity.

The core contribution of this research lies in its empirical analysis of ChatGPT’s performance across various software development tasks of increasing complexity. These tasks, categorized into four levels, “Easy”, “Medium”, “Hard”, and “Very Hard”, were designed to reflect real-world scenarios in diverse programming languages. For instance, “Easy” tasks include basic algorithms, while “Very Hard” tasks require contextual understanding and robust optimization techniques. Python was selected as the primary language for “Very

Hard” tasks due to its superior performance in earlier stages, highlighting the iterative refinement process applied during the experiments.

To address these challenges, standardized prompts were utilized to ensure consistency in evaluating ChatGPT’s capabilities. This approach facilitated the identification of system boundaries and informed the development of workarounds when limitations were encountered. Solutions were iteratively refined for each task until they met predefined test criteria, with successful approaches thoroughly documented. For example, tasks were considered complete for a programming language only when the generated solutions passed all test cases in the LeetCode test suite [28] and adhered to programming standards.

A comprehensive evaluation framework was adopted to assess ChatGPT’s effectiveness. This framework incorporated metrics such as accuracy, adaptability, efficiency, and solution quality across multiple dimensions, including maintainability, readability, and performance. Results are visually represented in tables, in Chapter 6, to provide clear insights into the model’s strengths and weaknesses. For instance, one finding highlights ChatGPT’s proficiency in Python but reveals limitations in handling languages like Erlang. Moreover, the analysis explores strategies to address system boundaries, such as rephrasing ambiguous prompts or introducing step-by-step guidance to enhance response accuracy.

Ethical considerations are critically examined, such as the potential risks of over-reliance on AI-generated solutions and the implications of using ChatGPT in collaborative environments. This ensures responsible usage of the technology and aligns the findings with broader discussions on ethical AI practices.

In conclusion, this thesis contributes to the ongoing exploration of AI in software engineering by providing empirical insights into ChatGPT’s practical applications, limitations, and areas for improvement. By combining performance evaluation with an experimental framework, the findings aim to bridge the gap between theoretical research and practical applications, helping developers better understand how to use large language models in real-world programming environments. The insights presented serve as a foundation for future advancements in integrating AI tools into the software development lifecycle.

1.1 Research Objectives and Experimental Methods

With the growing demand for natural language processing tools, evaluating their performance is essential. This thesis examines the capabilities of ChatGPT, focusing on its effectiveness in handling programming tasks and its potential to enhance software development processes, as outlined in the Research Question 1.

The research aims to:

- **Investigate Autonomy Potential:** Determine whether ChatGPT can autonomously handle programming tasks or should be used as a collaborative tool to improve efficiency in software development workflows.
- **Evaluate Performance:** Assess ChatGPT’s ability to solve diverse software engineering challenges by measuring solution accuracy, correctness, and efficiency, providing insights into its practical benefits.

- **Examine Proficiency Across Programming Languages:** Evaluate ChatGPT’s performance across Python, C, Java, Golang, and Erlang to identify language-specific strengths or biases. These languages were selected to represent a mix of paradigms, industry relevance, and complexity levels.
- **Perform Real-World Programming Challenges:** Beyond basic algorithmic tasks, assess ChatGPT’s ability to address real-world development scenarios, exploring its practical utility and limitations in professional environments.
- **Analyze Task Complexity Impact:** Examine ChatGPT’s adaptability to challenges of varying complexity, from basic algorithms to real-world scenarios, and analyze how complexity affects its performance.
- **Assess Information and Interaction Requirements:** Investigate the level of detail and interaction needed for ChatGPT to generate accurate responses, providing practical insights into its usability in software development.
- **Evaluate Consistency and Solution Quality:** Measure ChatGPT’s consistency by comparing outputs from repeated prompts across sessions and assess solution quality based on correctness, maintainability, and performance.
- **Identify System Limitations:** Explore scenarios where ChatGPT excels and where it struggles, identifying strategies to mitigate its limitations in software development tasks.

To address these objectives, we implemented a structured methodology:

- **Programming Capability Assessment:** ChatGPT was tasked with solving progressively complex programming challenges across Python, C, Java, Golang, and Erlang. These languages were chosen for their diverse paradigms and practical relevance. For each difficulty level, the programming language with the weakest performance was excluded from further analysis.
- **Incremental Challenge Provision:** Task descriptions were provided upfront or incrementally based on complexity, allowing us to evaluate ChatGPT’s ability to adapt to changing requirements.
- **Consistency and Reliability Testing:** Identical prompts were submitted multiple times across chat sessions, either by regenerating responses or starting new conversations. This approach identified response variations and assessed ChatGPT’s reliability.
- **System Boundary Exploration:** Ambiguous and incomplete prompts were introduced to determine the limits of ChatGPT’s functionality, offering insights into how it handles unclear inputs and where performance degrades.
- **Solution Evaluation and Testing:** Solutions were tested manually or using the LeetCode test suite [28] to ensure their correctness, efficiency, and applicability. This step confirmed compliance with programming standards.

- **Error Feedback and Iterative Improvement:** For solutions containing errors, detailed feedback, such as error messages and expected outputs, was provided to ChatGPT. This iterative process allowed us to assess its ability to learn from feedback and improve.
- **Comprehensive Solution Evaluation:** Solutions were evaluated for comprehensibility, maintainability, efficiency, and performance, ensuring a holistic assessment of ChatGPT's coding abilities.
- **Documentation and Analysis:** All interactions with ChatGPT were documented, including prompts, responses, and evaluations, ensuring reproducibility and providing a reference for future research.

Following this methodology, we systematically assessed ChatGPT's strengths and limitations across programming languages and complexity levels. This approach offered valuable insights into ChatGPT's potential as an auxiliary tool in software development. It highlighted areas for future improvement, ensuring a balanced evaluation of its capabilities and practical applications.

1.2 Requirements

This section outlines the requirements for integrating ChatGPT 3.5 into the study, establishing a foundational framework that ensures consistency and relevance in evaluating its suitability for software engineering tasks. A clear definition of system boundaries is critical to this experimental setup, ensuring a precise understanding of the scope and capabilities of ChatGPT 3.5 as provided by OpenAI.

ChatGPT 3.5 was selected for this research due to its accessibility, compatibility with the objectives, and ability to generate code aligned with industry standards. Listing 1.1 summarizes the specific capabilities leveraged in this study, including prompt-response interactions and multi-language support, making it particularly suitable for addressing programming challenges.

To ensure a realistic and standardized evaluation, the programming challenges and corresponding test suites were sourced from LeetCode, a widely recognized platform offering practical tasks aligned with real-world software engineering practices. LeetCode's automated test cases provide a consistent benchmark for assessing solution quality across multiple programming languages, ensuring fairness and transparency in evaluations. [28]

The primary objective of this study is to evaluate ChatGPT's ability to generate solutions to programming challenges within specified time constraints. Time-based metrics are complemented by code quality assessments, including adherence to best practices, correctness, and efficiency. These metrics are critical for understanding ChatGPT's performance in tasks requiring speed and accuracy.

By adopting a systematic and data-driven methodology, this research aims to provide actionable insights into the application of ChatGPT in software engineering, particularly regarding its capacity to meet the dynamic and complex demands of the field.

Chapter 2

Related Research

ChatGPT, a large-scale language model developed by OpenAI, has been the subject of numerous studies evaluating its performance in various aspects of software engineering, including code generation, program repair, and code summarizing. [32][49] Studies have used two benchmarks to compare ChatGPT's performance with state-of-the-art approaches. [49]

For code generation, they provided a natural language description of the desired code and evaluated ChatGPT's ability to generate it. This task is crucial in software development as it can significantly reduce the time and effort required to write code. The studies found that ChatGPT can generate accurate and efficient code based on the descriptions provided. [49][21] It has been evaluated for its ability to reduce the time and effort required in source code creation, simplifying the overall development process.[29] However, it has been observed that the quality of the code generated by ChatGPT can vary based on the complexity of the task and the level of detail provided in the description. [27]

For program repair, researchers introduced a bug into a code snippet and evaluated ChatGPT's ability to suggest a fix for the bug. Program repair is a critical aspect of software maintenance, and the ability of an AI model to accurately identify and fix bugs can greatly improve the efficiency of the software maintenance process. The studies found that ChatGPT's explanations of incorrect code provide valuable insights into the developer's original intentions, thus it can effectively suggest fixes for a variety of common bugs.[49] ChatGPT's performance is competitive with common deep learning approaches like Co-CoNut and Codex [45]. However, its repair performance drops significantly when the code is confusing or documentation is removed. [38]

Regarding code summarizing, the studies evaluated whether ChatGPT could produce consistent explanations for code samples with the same functionality. Code summarizing is essential for understanding the functionality of a code snippet, especially in large code bases or when working with code written by others. The study by Sun et al. [47], shows that ChatGPT's performance is significantly worse than all three state-of-the-art models, including NCS, CodeBERT, and CodeT5, regarding their BLEU and ROUGE-L scores. [47] However, other studies have found that ChatGPT can perform a human-like summarizing evaluation using four human evaluation methods on five datasets. [14] Furthermore, it has been found that ChatGPT can produce accurate and consistent summaries for a variety of code samples. [49]

Researchers highlight the potential of ChatGPT in automating various aspects of software engineering, while also pointing out the room for improvement. [17][13][55] They provide valuable insights into large-scale language models for programming assistance, demonstrating the importance of prompt engineering and providing a better understanding of ChatGPT's practical applications for software engineering. [49][8] One of the main limitations is its limited attention span. Detailed and comprehensive descriptions can restrict ChatGPT's focus, preventing it from leveraging its vast knowledge to solve the problem. This suggests that while ChatGPT can handle complex tasks, the presentation of them can significantly impact its performance [49].

2.1 Prompt Engineering with ChatGPT

In [52], the authors present a promising approach to prompt engineering with ChatGPT. They propose a catalog of prompt engineering techniques in pattern form for software engineering that classifies patterns according to the types of problems they solve. The prompt patterns have been applied successfully to improve the outputs of large language model conversations, including those related to software engineering, providing reusable solutions to common problems when conversing with large language models. Furthermore, they can solve common problems when using LLMs to automate software engineering activities, ensuring code is decoupled from third-party libraries and creating an API specification from a requirements list. Paper [53] for instance, explores several prompt patterns that have been applied to improve requirements elicitation, rapid prototyping, code quality, deployment, and testing. [52][53]

Prompt patterns are reusable prompt designs that can guide the behavior of LLMs in specific software engineering tasks. They are like software design patterns, which provide a reusable solution to common problems in software engineering. There are two types of prompt patterns - those that program the LLM to accomplish a task, and those that customize future interactions with the LLM by defining additional rules to follow or information to use when performing subsequent tasks. [53]

Prompt patterns are designed to solve problems in LLM interaction, such as ensuring that programs adhere to certain design principles or secure coding guidelines. They can be used to establish rules and constraints that improve software quality attributes, such as modularity or reusability when working with LLMs.

For example, one prompt pattern involves using the LLM to act as a system and guide the user through individual screens, similar to classic text-based adventure games. Another prompt pattern involves using the LLM to generate user stories based on missing requirements to make a task possible. [53]

White and colleagues [52] offer a framework documenting patterns for structuring prompts, which can be applied to automate software development tasks. The paper proposes the concept of fundamental contextual statements, which are written descriptions of the important ideas to communicate in a prompt to an LLM. The key ideas are presented to the user as a series of simple, but fundamental, statements. Additionally, the paper discusses fundamental contextual statements, templates, and placeholders to customize the outputs and interactions with an LLM to further improve the automation of software engineering-related tasks with ChatGPT. [52]

The framework consists of three main components:

1. **Prompt Engineering:** Design prompts that can be used to produce specific responses from ChatGPT. The paper provides several examples of prompt patterns that can be used for different software engineering tasks, such as generating code, writing documentation, and answering questions related to software development. [52]
2. **Model Training:** Fine-tune the model on a specific task or domain. The paper suggests that the model can be trained on a large corpus of text related to the task or domain, such as software documentation or code repositories. [52]
3. **Model Evaluation:** Evaluate the performance of ChatGPT on the specific task or domain. The paper suggests that the model can be evaluated using perplexity, accuracy, and F1 score, as metrics. [52]

Furthermore, they explain how prompts can be built from multiple patterns and illustrate prompt patterns that benefit from combination with other prompt patterns. [52]

In "Prompt Engineering with ChatGPT: A Guide for Academic Writers" [16], Giray discusses the significance of prompt engineering for academic writers and researchers. The author emphasizes that prompt engineering skills are becoming increasingly important, for academic writing, as artificial intelligence continues to advance. The paper provides a comprehensive guide on using LLMs like ChatGPT to enhance the writing process. It discusses the concepts of prompt engineering, large language models, and the techniques of prompt writing. The author suggests that academic writers can navigate the changing landscape and enhance their writing process, by acquiring prompt engineering skills. [16]

Liu and colleagues conduct an empirical study on prompt engineering with ChatGPT, in the paper "Jailbreaking ChatGPT via Prompt Engineering: An Empirical Study". [25] They discuss the challenges related to content constraints and the potential misuse of large language models like ChatGPT. The authors provide valuable insights into how prompt engineering can be used to overcome these challenges and unlock the full potential of ChatGPT. They suggest that prompt engineering can guide the behavior of ChatGPT, enabling it to generate more accurate and relevant responses. The authors also discuss the importance of evaluating the performance of ChatGPT and provide several metrics for evaluation, including perplexity, accuracy, and F1 score [25].

Furthermore, the paper "Prompt Engineering For ChatGPT: A Quick Guide To Techniques, Tips, And Best Practices " [10] provides a quick guide to techniques, tips, and strategies for effective prompt crafting. It discusses the importance of clarity, explicit constraints, experimentation, and leveraging different questions. The paper also provides several examples of how these techniques can be applied in practice to improve the effectiveness of prompts and enhance the performance of ChatGPT. The authors suggest that by following these techniques, users can craft more effective prompts that guide the behavior of ChatGPT and obtain more accurate and relevant responses [10].

In conclusion, prompt engineering with ChatGPT is a promising approach that can enhance the automation of various software engineering tasks. It offers a powerful tool to improve the performance of LLMs and automate diverse tasks. Prompt patterns, fundamental contextual statements, templates, and placeholders can help to customize the

outputs and interactions with an LLM. [53] However, it also presents challenges that need to be addressed to ensure the safe and effective use of these models. Future research is required, to develop more effective prompt engineering techniques, and address the challenges.

2.2 Software Engineering with ChatGPT

Paper [46] explores the utility of ChatGPT for common software engineering tasks such as code summarizing, test oracle generation, and code clone detection. The authors conducted experiments to test ChatGPT's performance on these tasks and compared its output to human expert output and state-of-the-art tool output.

They gathered ten random samples for each task and asked ChatGPT to perform the desired task. They initiated a conversation with ChatGPT with a specific software engineering task, evaluating its response by comparing it with human experts and the output from state-of-the-art tools. For example, in [46], the authors provided a Java method to ChatGPT and requested it to provide a code review of the method. They then compared the provided review with the human expert review. In another experiment, they performed code clone detection with ChatGPT by asking if the provided code snippets were duplicates or clones of each other. [46]

Several other examples of how ChatGPT can be used for software engineering tasks are provided by paper [52]. One example is the use of ChatGPT to generate Python scripts for automating the deployment of software applications. The authors explain how ChatGPT can be prompted to ask the user questions about their software application until it has enough information to generate a Python script that automates deployment.

Another example is the use of ChatGPT to generate code for software engineering tasks. White et al. [52] explain how ChatGPT can write a Boolean query for a systematic review literature search. They also discuss how ChatGPT can be used for conversational automated program repair and to simulate a Linux terminal window. [52]

The researcher's results suggest that ChatGPT performs adequately for many software engineering tasks and often provides detailed responses better than human experts or state-of-the-art outputs. Nonetheless, ChatGPT in its present form provides incorrect answers for complex tasks. For example, in [46], Sridhara and colleagues found that ChatGPT performed well for code summarizing, test oracle generation, and code clone detection tasks. However, the performance was mixed for code review generation, with agreement between ChatGPT's generated code review and the original developer's code review in only four out of ten cases. This shows that ChatGPT's performance can be inconsistent across software engineering tasks. Additionally, ChatGPT's responses are sometimes too general and lack specificity, which can be problematic for software engineering tasks that require precise and detailed answers. Moreover, ChatGPT's responses are influenced by its training data, which can lead to biased or incorrect responses. Finally, ChatGPT's performance is limited by the quality of the input it receives. [46][52]

Another interesting approach of using ChatGPT as a tool in software development is explored in the master's thesis by Adam Hörnemalm. The study aimed to answer three research questions: Is ChatGPT an effective tool? Are there risks associated with using AI tools? What can be done to improve the developer experience of using AI tools? [20]

The thesis was conducted in two phases: an initial exploration phase where five senior-level developers were interviewed to form a theoretical foundation for software development, and a data collection phase. In the data collection phase of the study, participants were given a set of representative tasks to complete using ChatGPT. These tasks were based on the theoretical foundation of software development and aimed to serve as representative tasks that software developers must do in their day-to-day work. One example of a task given to participants was to “write a function that takes a list of integers as input and returns the sum of all even numbers in the list”. Participants were asked to use ChatGPT to help them complete this task. Another example was to “write a function that takes a string as input and returns the number of vowels in the string”. Again, participants were asked to use ChatGPT to help them complete this task. [20]

The main findings show that ChatGPT can make software developers more efficient during coding and planning-related tasks but has trouble with communication-based tasks due to its unintuitive way of formatting text. Furthermore, they found that junior developers trusted and relied more on the answers given by ChatGPT, which could be a potential risk. Despite this, ChatGPT enabled every participant to solve the tasks, regardless of experience level. [20]

Bera et al. [6] investigated a unique example of using ChatGPT in SE. They suggest using ChatGPT as a virtual development team member to inform, coach, and execute a share of the development work. The study argues that ChatGPT could provide relevant generic knowledge on agile methods, give tailored advice for a development project based on contextual information, or even perform work according to agile practices themselves based on provided project data. The authors performed a small empirical experiment to assess the performance of ChatGPT when given some tasks that would typically be performed by an agile coach or scrum master. They provided ChatGPT with user stories and asked it to identify the most important ones, estimate their complexity, and assign them to the appropriate sprint. Bera and colleagues [6] found that ChatGPT performed these tasks with reasonable accuracy. However, using such a tool is not trivial, and the paper discusses some potential issues and boundaries that ChatGPT may face in supporting agile software development. These include the need to investigate the true capacity of ChatGPT to provide valid information, understand contexts, and apply techniques correctly, as well as the uncertainty of how a virtual team member like ChatGPT would be perceived and integrated by its human colleagues. [6]

Bera et al. [6] conclude that while there is still a lack of understanding about how to apply agile practices in the public sector cultural change is necessary for successful implementation. They emphasize the importance of agile practices training and strict role and responsibility definitions to address these challenges. [6]

In [35], the authors conducted an empirical investigation comparing the performance of software engineers and AI systems, specifically ChatGPT, in solving coding problems, similar to the focus of our research topic. Nascimento and colleagues [35] used ChatGPT to generate code and formulate software testing strategies for developers and testers. They compared ChatGPT-generated code to code written by developers who uploaded it on Leetcode, a platform for practicing coding skills. [35][28]

They selected a contest from Leetcode that contains programming problems at different difficulty levels and categorized the programmers according to the number of programming contests they have already attended in Leetcode. [28]

The authors validated three alternative hypotheses while three were rejected:

1. For “Easy” and “Medium”-level problems, ChatGPT enhances the performance of coding solutions compared to those provided by novice contest programmers.
2. For “Easy” and “Hard”-level coding problems, ChatGPT demonstrates superior memory efficiency compared to the solutions offered by either novice or experienced contest programmers.
3. There is no evidence showing that ChatGPT surpasses the performance of solutions provided by experienced contest programmers.

The solutions generated by ChatGPT-4 surpass a significant percentage of existing solutions for each problem. Regarding memory efficiency, the solutions given by ChatGPT outperform a considerable percentage of others. However, this comparison includes all previously provided solutions, thus lacking clarity on how these automated solutions perform relative to software engineers with varying expertise levels. The researchers address this by exclusively considering the solutions provided by their selected participants to the “Easy”, “Medium”, and “Hard” problems. [35]

Nascimento and colleagues [35] conclude that ChatGPT demonstrated superior performance over novice contest programmers in solving “Easy” and “Medium”-level problems and exhibited better memory efficiency in one of the “Medium”-level problems. However, there was no evidence to suggest that ChatGPT surpassed the performance of solutions provided by experienced contest programmers. Additionally, the paper emphasizes the importance of considering non-functional requirements such as energy efficiency, vulnerability, fairness, and safety when evaluating AI-based solutions in software engineering tasks. [35]

Most studies conclude that ChatGPT can be used to help with simple software engineering tasks while highlighting the need for further research and development to improve ChatGPT’s performance for certain tasks.

2.3 Debugging with ChatGPT

In [12] and [48], the authors suggest that ChatGPT can be used for debugging, software testing, natural language processing, and documentation generation. ChatGPT can provide suggestions and corrections to code based on its understanding of the relationships between code and bugs. The suggestions can be based on its prior knowledge of programming languages, common bug patterns, and best practices in software development. Using ChatGPT for these tasks, developers can save time, improve efficiency and effectiveness in software development, enhance collaboration and knowledge sharing, and improve the software documentation’s quality. [12][48]

Nigar and colleagues [48] examine the characteristics of ChatGPT and how they can be used for debugging assistance, bug prediction, and bug explanation to help solve programming problems. The authors suggest using ChatGPT to provide debugging assistance for programming codes leveraging its natural language processing (NLP) capabilities, knowledge representation, and pattern recognition abilities. This involves training the model on a large dataset of code snippets, bug reports, and related information about software development. The goal is to enable the model to understand the relationships between code

and bugs, and to use this understanding to identify and fix bugs in new code snippets. Once the model has been trained, it can provide suggestions and corrections to code based on its understanding of the relationships between code and bugs. In [48], the authors mention that ChatGPT’s natural language generation capabilities make it “Easy” for developers to understand its results compared to traditional debugging tools, which can be more complex to use. [48]

The paper also highlights the importance of using other debugging tools and techniques to validate ChatGPT’s predictions and explanations. ChatGPT may not be able to identify all types of bugs, especially those that are complex or rare. It can provide incorrect suggestions or corrections, which need validation using other debugging tools and techniques. Additionally, it may be unable to explain its suggestions or corrections in an understandable way to developers unfamiliar with natural language processing. It may require vast computational resources and time to train and use effectively. [48] In [12], the authors suggest that researchers should develop methods to

- Detect and mitigate bias in ChatGPT models.
- Protect the privacy and security of user data.
- Improve the transparency, data availability, and quality of ChatGPT models.
- Optimize the computational efficiency of ChatGPT models. [12]

The study concludes, by highlighting the potential of ChatGPT as one part of a comprehensive debugging toolkit, and the possibility of various benefits of combining ChatGPT’s strengths with those of other debugging tools to identify and fix bugs more effectively. [12][48] Furthermore, it discusses the ethical concerns associated with ChatGPT and other AI models, such as bias, privacy, accountability, and transparency. [12]

In addition to the studies by Fraiwan et al. [12] and Nigar et al. [48], other researchers have explored the use of ChatGPT for error handling and debugging. For instance, Xia and Zhang [54] conducted a study titled “Keep the Conversation Going: Fixing 162 out of 337 bugs for \$0.42 each using ChatGPT”, discussing the potential of ChatGPT in identifying and fixing programming errors. They demonstrated that ChatGPT could fix bugs at a low cost, further emphasizing the potential as a cost-effective debugging tool. [54]

Xia and Zhang [54] highlighted the importance of continuous conversation with ChatGPT during the debugging process. They found that continuous interaction with ChatGPT during the debugging process allows the model to understand the code context and bug nature, leading to more accurate and effective bug fixes. This approach contrasts with traditional debugging tools, which often require developers to manually input information about the bug and the code, potentially leading to errors or deletions that can affect the accuracy of the debugging process. [54]

Moreover, Lou et al. [30] in their paper “No more manual tests? Evaluating and improving ChatGPT for unit test generation” [30], explored the use of ChatGPT for generating unit tests. They found that ChatGPT could generate effective unit tests that could help identify bugs. This suggests that ChatGPT could be used to fix bugs and prevent them from occurring in the first place.

However, like Fraiwan et al. [12] and Nigar et al. [48], Xia et al. [54] and Lou et al. [30] emphasized the need for caution when using ChatGPT for debugging. They noted that while ChatGPT can provide valuable assistance in the debugging process, it should not replace traditional debugging tools and techniques. Instead, it should be used as a complementary tool to enhance the effectiveness of the debugging process.

Another noteworthy study is by Pu et al. [40], titled “A process-algebraic approach to workflow specification and refinement” [40]. In this paper, the authors propose a process-algebraic approach to error handling in programming. They argue that this approach can provide a more systematic and formal way to handle errors, leading to more reliable and robust software. This approach involves defining a set of algebraic operations that can be used to specify and refine workflows, and then using these operations to handle errors systematically and formally. [40]

Gibbons and Oliveira [15] explore iterator patterns in error handling. They suggest that the Iterator pattern can provide a more intuitive and efficient way to handle errors in programming, especially in the context of data structures and algorithms. The Iterator pattern involves defining a standard interface for traversing a collection of items, and then using this interface to handle errors systematically and efficiently. [15]

In conclusion, these studies provide valuable insights into different approaches to error handling in programming, and how they can be integrated with ChatGPT to enhance its debugging capabilities. While current studies offer crucial perspectives on the potential benefits and challenges of using ChatGPT for debugging, further research is necessary to fully comprehend its capabilities and limitations in this context. As AI models like ChatGPT continue to evolve, their role in debugging will likely become increasingly important.

2.4 Ethical aspects when using ChatGPT

ChatGPT, with its natural language interactions, has the potential to enhance SE research practices by providing efficient and accessible information analysis and synthesis. However, it also introduces ethical challenges that need careful consideration.

Muhammad Azeem Akbar et al. [3], highlights the ethical principles and identifies motivators and demotivators of using ChatGPT in SE research. The motivators include generating synthetic data for software testing, fine-tuning specific SE domains, and enhancing the efficiency of knowledge extraction and collaboration. The demotivators include ethical concerns such as plagiarism, privacy, data security, and the risk of generating biased or harmful data. Additionally, there are technical challenges, including difficulties in integrating ChatGPT into existing workflows, misalignment with stakeholder preferences and expectations, and the need for significant manual editing or correction [3]

ChatGPT can generate text that closely resembles existing content. Researchers must be cautious to avoid unintentional plagiarism when using ChatGPT for writing papers, documentation, or code comments. For instance, if ChatGPT generates a code snippet resembling an open-source project, acknowledging the source is essential. [3]

Furthermore, users may input sensitive information or data. Ensuring privacy and data security is crucial. Researchers should be aware of the risks associated with sharing confidential data with an AI model and take necessary precautions, for example using techniques like differential privacy to protect user information. [3]

Integrating ChatGPT into existing workflows can be technically challenging. Researchers may encounter issues related to API integration, deployment, and scalability. Ensuring seamless integration requires careful planning and technical expertise.

Moreover, stakeholders, including users, clients, and colleagues, may have varying expectations regarding ChatGPT's capabilities and limitations. Users must manage these expectations and communicate transparently about ChatGPT's role and limitations. [3]

While ChatGPT can assist in generating content, manual editing and correction are often necessary. Post-processing is necessary to refine ChatGPT-generated output. To address these challenges, Akbar et al. conducted a questionnaire-based survey to evaluate the identified elements. The researchers developed decision models using Interpretive Structure Modeling (ISM) and Cross-Impact Matrix Multiplication Applied to Classification (MICMAC) approaches to analyze the relationships between the ethical principles of using ChatGPT in SE research. [3]

ISM helps analyze the relationships between ethical principles and ChatGPT usage in SE research. By understanding these relationships, researchers can devise effective strategies. For example, if privacy is a concern, ISM can help make decisions regarding data handling and sharing. [3]

MICMAC identifies clusters of factors that influence ethical integration. Researchers can prioritize actions based on these clusters. For instance, addressing biases, a critical ethical concern, can be a high-priority action. [3]

The resulting decision models aim to balance the potential benefits of ChatGPT with the ethical risks and help researchers devise effective strategies for ethically integrating ChatGPT into their research. [3]

In addition to the ethical principles identified by Akbar et al. [3], other aspects need consideration when using ChatGPT in software engineering. One is the issue of fairness, which is a major concern when using AI tools like ChatGPT, as highlighted in the paper on "Security and Ethical Considerations When Using ChatGPT" [50] The training language models may unintentionally reinforce biases, and they should adopt ethical practices, such as continuous evaluation and data collection that are representative and diverse. [50]

Transparency is another ethical aspect that needs to be considered. As AI models become more complex, their decision-making process becomes unclear. This is referred to as the "black box" problem. This makes it difficult for users to understand how a particular output was generated. This lack of transparency can lead to mistrust and misuse of the tool. For example, if a software engineer relies on ChatGPT for code generation but doesn't understand how the tool generates the code, they might blindly trust the output without thoroughly reviewing it. This could lead to the propagation of bugs or security vulnerabilities in the software. Therefore, future development should focus on improving the transparency of ChatGPT, such as providing clear documentation and explanations of the tool's capabilities and limitations. [11]

The paper [50] also discusses the potential security risks associated with using ChatGPT in software engineering. One of the key points raised in this paper is sharing proprietary information with the AI tool. For instance, when software engineers use ChatGPT to check their programming code, they might unintentionally share confidential data with the tool. This could potentially lead to violating data privacy and security regulations, a significant ethical concern. Therefore, users must be cautious and ensure they do not share sensitive information when interacting with ChatGPT [50].

Lastly, the issue of accountability is a critical ethical aspect when using ChatGPT in SE. If ChatGPT generates incorrect or harmful outputs, who should be held responsible? Is it the tool's developers, the users, or the organization using it? These questions need to be addressed to ensure clear accountability. For instance, if ChatGPT is used for automated code generation and the generated code causes a system failure, who should be held accountable for the failure? The developers of ChatGPT, the software engineer who used the tool, or the organization that deployed the software? [11][50]

In conclusion, while ChatGPT can offer many benefits in SE, it is crucial to consider these ethical aspects when using this tool. By being aware of these ethical concerns, such as data bias and fairness, or privacy and security, and taking steps to address them, we can ensure that we use ChatGPT responsibly and effectively in SE [52][12][11].

2.5 Conclusion

The analysis of related research regarding ChatGPT in software engineering, as outlined in this chapter, provides a comprehensive overview of its capabilities and limitations. The empirical evidence describes ChatGPT's role as a transformative tool in code-to-text and text-to-text transformations, highlighting OpenAI's advances in effectively aligning the model to user intentions. This enhances the model's ability to understand and generate human-like responses. The results show that ChatGPT can be a valuable asset for software engineers, particularly for tasks involving complex natural language processing requirements [46].

Studies on code generation, program repair, and code summarizing establish that ChatGPT can significantly reduce the time and effort required in development processes. For instance, ChatGPT's capability to generate accurate and efficient code from natural language descriptions illustrates its potential to streamline development, effectively reducing the cognitive load on developers. [49][21] The model's ability to propose fixes for bugs in code snippets highlights its utility in software maintenance, a critical aspect of the software development life cycle. [49]

The quality of ChatGPT's responses can be inconsistent, particularly with complex tasks or descriptions that are not detailed. This variability highlights the necessity for further improvement. [27] Additionally, ethical considerations such as fairness, transparency, and accountability must be addressed to ensure the responsible utilization of ChatGPT. The studies highlight the importance of continuous evaluation to mitigate biases and improve the trustworthiness of AI tools like ChatGPT. [50]

The research indicates a critical need for dedicated efforts in research and development to enhance ChatGPT's effectiveness. These efforts should focus on improving the technical aspects of the model. Addressing these challenges requires a combined approach involving technical improvements, regulatory oversight, and ethical considerations to ensure AI advancements positively impact software engineering.

In conclusion, the integration of ChatGPT in software engineering holds promise. Future research and development initiatives must focus on these areas, aiming to develop a robust, ethical, and efficient tool that meets the evolving needs of software engineers. Collaboration among AI researchers and software engineers will be crucial in maximizing ChatGPT's potential as a foundational technology in software engineering. [46]

Chapter 3

ChatGPT

ChatGPT (Chat Generative Pre-trained Transformer) is a chatbot enhanced with machine learning capabilities, developed by OpenAI. OpenAI, founded in 2015, is an AI research organization established by prominent technology leaders, including Elon Musk and Sam Altman. OpenAI consists of two entities - OpenAI LP, a for-profit corporation, and OpenAI Inc, a non-profit organization.[3] ChatGPT 3.5, which we use in this research, utilizes the GPT-3 family of large language models. [12][46][35]

ChatGPT is based on the Transformer architecture, a neural network architecture. This architecture is designed to process sequential data, such as natural language, and is highly effective in diverse natural language processing tasks. [2]

It is an LLM using deep learning techniques to generate human-like text. [12][53] ChatGPT 3.5 has been developed to provide real-time communication in response to user's requests. [36] It has received widespread attention for its articulate responses across diverse knowledge domains and powerful capabilities, various possible applications, and the great possibility for abuse. [12][46]

To develop ChatGPT, OpenAI used a large-scale unsupervised learning approach, where the model was trained on a vast amount of text data without any explicit supervision. Training data was sourced from various publicly available sources, including books, articles, and websites. [3][2]

During the training process, the model learns to predict the next word in a sequence of text, given the previous words. This allows the model to learn patterns and relationships between words, thus generating coherent and contextually relevant responses to natural language inputs. [52][12][1][2] It is further improved with supervised and reinforcement learning techniques to enhance its performance in specific domains. [46]

ChatGPT has been trained on various tasks, making it a versatile tool for natural language processing tasks and software engineering. Its application in software engineering has gained significant attention due to its potential to streamline various tasks and improve productivity. One of the critical aspects of using ChatGPT in software engineering is prompt engineering. Prompt engineering involves crafting inputs that guide the model to generate useful and relevant outputs. This is especially important in software engineering, where precision and accuracy are crucial.

Prompt engineering can be used in code generation, debugging, and documentation. According to a study by Ahmad et al., ChatGPT can assist in software architecture by providing recommendations and generating code structures. This collaborative approach

between human developers and AI can enhance the efficiency of software processes. [2]

Moreover, White et al. highlight the use of ChatGPT to improve code quality and refactoring. By using well-crafted prompts, developers can instruct ChatGPT to review and suggest improvements for existing code, identify potential bugs, and ensure adherence to coding standards. This saves time and helps maintain high-quality software. [53]

ChatGPT can generate code, reducing the effort required for repetitive coding tasks. Using specific prompts related to the desired functionality, developers can obtain code snippets that can be integrated into larger projects. This capability can be beneficial for setting up APIs, creating unit tests, and generating configuration files. [24]

Documentation is another crucial aspect of software engineering that is often overlooked due to time constraints. ChatGPT can assist in generating detailed documentation by transforming technical specifications into comprehensive documentation. In addition, it can answer questions related to code, making it a valuable tool for knowledge sharing among development teams. [24]

Debugging is another area where ChatGPT can assist. By describing the issue and providing relevant code snippets, developers can use ChatGPT to identify potential causes of bugs and receive suggestions for fixes. This accelerates the debugging process and helps resolve issues more efficiently. [37][24]

The integration of ChatGPT into software engineering workflows offers several benefits, including increased productivity, improved code quality, and enhanced collaboration. However, there are also challenges to consider. One of the primary concerns is the accuracy and reliability of the outputs generated. While ChatGPT can produce high-quality text, the generated responses can be incorrect. Therefore, human oversight is essential to validate the generated outputs. [37]

Another challenge is the ethical implications of using AI in software engineering. Akbar et al. highlighted that using AI models such as ChatGPT raises questions about accountability, transparency, and the potential for misuse. Guidelines and best practices must be established for the ethical use of AI in software development. [37][3]

With a thorough understanding of ChatGPT's capabilities, potential applications, and limitations, we now focus on analyzing various use cases where it can be integrated into software engineering workflows.

3.1 Use-Cases and Benefits

ChatGPT demonstrates significant potential in software engineering, particularly for tasks requiring natural language processing (NLP) capabilities, such as translation, summarization, and text completion. Leveraging advanced attention mechanisms, ChatGPT tailors outputs to specific inputs, enhancing its contextual relevance and utility in software development workflows. [53][42]

In software engineering, ChatGPT provides support for code generation, bug resolution, code completion, and software testing strategies. It also aids in synthetic test data generation, code summarization, software maintenance, documentation, and project organization. These applications underscore ChatGPT's adaptability and its role in streamlining workflows. [35][3][20][1]

The following consolidated overview highlights ChatGPT's use cases and their corresponding benefits:

- **Code generation and completion:** ChatGPT can serve as a valuable tool for automated code generation and completion, enabling developers to build software systems efficiently while minimizing human errors. It can generate code snippets or entire programs based on user-provided descriptions, significantly reducing development time. For instance, studies such as those by Beganovic et al. [1] and White et al. [53] illustrate ChatGPT's effectiveness in generating Python code snippets and completing code based on functionality specifications. [12][53][51][1]
- **Code documentation and commenting:** ChatGPT can support the creation of comprehensive and user-friendly documentation. By processing provided information, ChatGPT generates detailed explanations of software components, improving code readability and knowledge transfer among team members. Its capability to write descriptive comments further enhances code understandability, fostering collaboration and quality assurance. [35][42][12][4]
- **Debugging and bug resolution:** ChatGPT can aid in identifying and resolving code issues by analyzing natural language problem descriptions and code snippets. Studies, such as those by Fraiwan and Khasawneh [12] or Beganovic et al. [1], highlight its use as a debugging assistant, providing explanations and solutions. Haque and Li [18] also demonstrate ChatGPT's ability to simplify the debugging process through immediate feedback and recommendations. [18][12][1]
- **Software testing and test case generation:** ChatGPT can contribute to test case creation and synthetic test data generation, helping developers evaluate software functionality and performance. By automating these processes, it ensures comprehensive test coverage, including edge cases, and enhances software quality. [12][53]
- **Data dependency detection:** Utilizing its analytical capabilities, ChatGPT can identify relationships between variables within code, aiding in data dependency detection, a critical task in code analysis and optimization. Ma et al. [33] demonstrate its utility in understanding data dependencies for enhanced code analysis.
- **Requirements elicitation and software refactoring:** ChatGPT can assist in eliciting software requirements by generating user stories to address missing specifications. This can help developers better understand the necessary functionalities to complete a task. Additionally, it can suggest refactoring strategies for improving code structure and design patterns and best practices. [53]
- **Programming numerical methods:** For numerical computation tasks, ChatGPT can generate code to solve differential equations or optimization problems. Beganovic et al. [1] found it capable of producing promising solutions, though validation and refinement remain necessary for complex scenarios.
- **Test oracle generation:** ChatGPT can be used to create test oracles, providing expected outputs for given inputs. This enables the evaluation of software system behavior and functionality, improving the effectiveness of testing processes. [53]

Beyond its technical applications, ChatGPT offers advantages that enhance productivity, collaboration, and learning in software engineering:

- **Improved productivity and quality:** ChatGPT can suggest improvements to code quality and performance, such as refactoring techniques, adopting APIs, or enhancing visualization. These recommendations streamline workflows and reduce manual effort. [9]
- **Enhanced collaboration:** The model supports knowledge sharing by generating clear documentation and explanations, aiding team communication and reducing onboarding efforts. [42][12][4]
- **Educational support:** ChatGPT can offer personalized learning experiences by generating educational resources, answering questions, and creating adaptive learning pathways. This supports skill development and facilitates deeper understanding. [42][5][12]
- **Software maintenance and evolution:** By providing insights into code updates and adaptations, ChatGPT simplifies software maintenance and evolution. It can give insights into updating code components, migrating to new technologies, or adapting the software to new requirements. [53]
- **Accessibility:** ChatGPT enables natural language programming, as highlighted by Qin et al. [41], making coding accessible to individuals with diverse skill levels by allowing programming in plain text instead of traditional syntax. [41]

Despite its advantages, ChatGPT has limitations, including potential biases, occasional inaccuracies, and the need for rigorous evaluation of its outputs. Ethical considerations, such as those discussed by Hariri [19] and Ahmad et al. [2], are essential to ensure responsible integration of AI in software engineering workflows. Addressing these challenges fosters the effective and ethical use of ChatGPT's capabilities.

3.2 Issues and Boundaries

Researchers have found that ChatGPT performs credibly in many tasks and provides detailed and often better responses than human experts. However, it is important to acknowledge the limitations and challenges associated with its usage. [12][20] ChatGPT's responses are sometimes too general and lack specificity, which can be problematic for tasks requiring specific answers. Additionally, its performance is limited by the quality of the input it receives. More precisely, the quality of ChatGPT's response corresponds to the input given by the user. [46] This behavior of ChatGPT can be due to its limited training data, interpretability, creativity, and domain-specific knowledge. [1]

The following list provides an overview of ChatGPT's underlying issues and limitations.

- **Data Bias and Fairness:** One of the challenges with ChatGPT is the potential for inheriting biases from the training data. A limitation in the model's performance is present when trained primarily on a dataset biased towards a specific programming language or development methodology. This can result in inferior performance in

other languages or methodologies. Addressing data bias and ensuring fairness in the training process is important. This can be achieved using specific prompt patterns to enhance ChatGPT's fairness and reliability. [52][12]

- **Interpretability:** ChatGPT's lack of transparency, often referred to as a "black box" model, presents challenges in interpreting the provided results. This limits the ability to assess the validity and accuracy of its responses, especially for tasks that require adherence to specific design principles or coding standards. [46]
- **Limited Context:** ChatGPT operates within a limited context window, which impacts its ability to identify dependencies in software. Consequently, certain software engineering tasks that require a deep understanding of code dependencies, such as code refactoring or optimization, can be challenging for ChatGPT. [49]
- **Privacy and Security:** Generating sensitive information with ChatGPT poses privacy and security risks. The model can generate project-specific and sensitive data, which, when made public, can compromise privacy or security. [12]
- **Maximum token limitation:** ChatGPT, as a transformer-based model, has a maximum token limitation on its responses. This limitation may result in incomplete responses, particularly in cases where the generated code is truncated. This truncation of the generated code can lead to incomplete and unusable. [33][42]
- **Response Variation:** ChatGPT may show response variation for the same queries, which can impact consistency in the software development process. Variations in recommendations and responses can affect the synthesis and evaluation of software architecture. [2]
- **Debugging/Modifying Generated Code:** While ChatGPT can generate code, debugging or modifying the output may be challenging, especially when dealing with complex or rare bugs. The effectiveness of debugging with ChatGPT depends on the quality of the training data and the system's design. [48][53]
- **Reliability of Solutions:** There is a potential risk associated with developers heavily relying on ChatGPT's responses. The reliability of ChatGPT's solutions should be carefully considered. It is important to conduct comprehensive evaluations and comparisons of AI-based methods to understand the suitability of AI-based solutions. Hornemalm [20] and Nascimento et al. [35] emphasize the importance of evaluating AI-generated solutions in software engineering to ensure the reliability of the project. [20][35]
- **Format and Project Specificity:** ChatGPT's responses may not always conform to the specific format or structure required by certain software development tools or platforms, nor align with the preferences or expectations of stakeholders. This can lead to technical difficulties during integration into existing workflows, and disagreements and conflicts among project team members and stakeholders, potentially impacting project progress and success. [33][3]

- **Limited Understanding of Code Semantics:** While ChatGPT demonstrates a solid understanding of code syntax, it fundamentally lacks an understanding of code semantics, especially concerning dynamic semantics. There is practically no evidence that ChatGPT comprehends code semantics meaningfully. This absence of semantic understanding significantly limits its ability to execute tasks requiring an interpretation of the intended functionality, behavior, and execution flow of the code. [32]
- **Generic Output and Need for Refactoring:** ChatGPT tends to provide generic advice that may not fully address the specific needs of a development project. As a result, its responses often require significant manual editing or refactoring, which can diminish the efficiency gains that automation would otherwise offer. [6][3]
- **Technical Constraints:** Responses generated by ChatGPT can conflict with the technical constraints of the software development environment. This mismatch can lead to ineffective recommendations that are difficult to implement in the given technical context. [3]
- **Hallucination:** ChatGPT can generate incorrect or entirely fabricated information, even when it appears plausible. This poses challenges in software engineering, where precision is critical. For instance, the model might produce code snippets with nonexistent APIs, functions or libraries that do not exist, or incorrectly describe the behavior of a library function. Such inaccuracies can lead to errors or inefficiencies, which are critical concerns in software engineering. [23][31]
- **Non-determinism:** ChatGPT's responses can vary for identical prompts, making it difficult to reproduce results consistently. This unpredictability complicates tasks like debugging or refining code iteratively. For example, one response might follow a specific approach, while the next suggests a completely different method, increasing the time and effort needed for validation and integration. [39]

While ChatGPT has the potential to be a useful tool to automate many software engineering tasks, it is crucial to consider and address these limitations. By understanding these issues, user can evaluate and check the responses before using them, thus effectively integrating ChatGPT into their software engineering workflows.

3.3 Summary

In conclusion, ChatGPT, a powerful language model (LLM), has the potential to revolutionize software engineering. [52][12] As highlighted by the referenced studies [52, 12, 3], ChatGPT can significantly improve efficiency in software engineering by automating repetitive tasks. ChatGPT's ability to assist in code generation, debugging, and documentation can greatly benefit software engineers. By using well-crafted prompts, developers can get code snippets that integrate into larger projects. However, it is crucial to be cautious when relying on ChatGPT's responses. As noted in the previous section, human oversight is essential to validate and refine the AI-generated outputs, ensuring the result's accuracy, availability, and completeness. [48]

While ChatGPT offers numerous benefits, it is important to acknowledge and address the limitations. These include potential biases inherited from training data, lack of interpretability, and limited context. These issues need to be mitigated to ensure the reliability and fairness of ChatGPT's responses in software engineering tasks. These include the need for manual editing and correction, response variation, and potential difficulties in debugging and modifying the generated code. Furthermore, the ethical considerations of using ChatGPT in real-world scenarios cannot be overlooked. Ensuring the technology's responsible deployment and alignment with ethical guidelines is essential. [12][42]

Looking ahead, the potential of ChatGPT in software engineering is vast. We can expect to see even more innovative applications and use cases, as it continuously evolves. With its real-time communication capabilities and assistance in various software engineering tasks, ChatGPT has the potential to revolutionize our daily lives by incorporating AI-generated information. [36] However, to fully utilize the potential of ChatGPT in software engineering, further research is needed to explore its impact, limitations, and ethical concerns. It is important to continuously evaluate and refine AI-based solutions, enhancing reliability and understanding their suitability in software engineering workflows. Future research may focus on improving the accuracy of AI-generated code, enhancing the model's understanding of complex programming concepts, and developing more sophisticated prompt engineering techniques. Additionally, integrating ChatGPT with other AI tools and development environments could further streamline the software engineering process and foster innovation. [34]

In conclusion, ChatGPT represents significant advancements in natural language processing and can transform various aspects of software engineering. Its capabilities while considering its limitations and ethical implications, can be beneficial for diverse software engineering tasks. As ChatGPT continues to evolve, it is expected to find even more innovative applications and contribute to advancements in software engineering.

Chapter 4

Experiments

This chapter shifts focus from theoretical foundations to the practical aspects of our study, presenting a detailed exploration of programming assignments designed to evaluate ChatGPT’s capabilities in solving coding challenges.

The primary objective is to assess ChatGPT 3.5’s ability to interpret and solve practical coding problems, exploring its potential applications in software engineering while identifying its adaptability and limitations.

The tasks, ranging from algorithmic puzzles to real-world applications, are designed to test the boundaries of ChatGPT’s understanding. Each task description includes the necessary details to comprehend the challenges and requirements.

Task difficulty is divided into four categories: “Easy”, “Medium”, “Hard”, and “Very Hard”. These classifications are primarily based on LeetCode’s established metrics, except for the “Very Hard” category, which includes complex coding projects rather than isolated snippets. Additionally, the task descriptions for all categories, except for the “Very Hard” tasks, were sourced from LeetCode and are presented in the following subsections. Table 4.1 provides an overview of the task distribution across the difficulties. [28]

LeetCode is a widely recognized online platform designed to help individuals enhance their programming skills and tackle algorithmic challenges. It offers a vast repository of coding problems, ranging from “Easy” to “Hard”, to accommodate various levels of expertise. Specifically, it covers data structures, algorithms, and system design, making it a versatile tool for developing and testing coding skills relevant to software engineering. Additionally, LeetCode provides an integrated code editor that supports multiple programming languages and includes a test suite to evaluate submitted solutions against predefined test cases. This feature delivers immediate feedback on both correctness and efficiency. Moreover, LeetCode assesses solutions based on runtime and memory usage, enabling users to optimize their code for better performance. In the context of this thesis, LeetCode serves as a standardized platform to evaluate ChatGPT’s effectiveness in solving programming tasks. [28] Its key advantages include:

- A set of problems to test ChatGPT’s performance across various challenges.
- Automated test suite to objectively verify the correctness of ChatGPT’s solutions.
- Performance metrics, such as runtime and memory usage, to assess the efficiency of solutions.

By leveraging LeetCode, this experiment gains credibility using a well-established and widely respected platform in the programming and software engineering communities. [28]

For a systematic evaluation, we used five programming languages, Python, C, Java, Golang, and Erlang, for “Easy” tasks. As we progressed to “Medium” and “Hard” tasks, the number of languages was reduced, focusing on those with the best performance in earlier levels. The “Very Hard” tasks were addressed exclusively in Python, leveraging its superior results and ChatGPT’s integrated interpreter capabilities.

Time constraints played a significant role in simulating real-world conditions. “Easy” tasks had a limit of one hour, “Medium” tasks five hours, and “Hard” tasks up to ten hours. The “Very Hard” tasks, approached in a pair-programming style, had no defined time limit, as the focus shifted to evaluating ChatGPT’s collaborative potential.

Time constraints played a significant role in simulating real-world conditions. For “Easy” tasks, we set a time constraint of one hour for ChatGPT, as they are mostly straightforward algorithmic challenges. For “Medium” tasks, the time limit increases to five hours. For “Hard” tasks, ChatGPT has up to ten hours to solve each task in every programming language. These time constraints simulate realistic programming environments, assessing ChatGPT’s efficiency and problem-solving speed across various complexity levels. The “Very Hard” tasks, approached in a pair-programming style, had no defined time limit, as the focus shifted to evaluating ChatGPT’s collaborative potential.

The results of this study are presented in Chapter 6 with comprehensive tables, such as the following for the “Easy” tasks, illustrating ChatGPT’s performance:

- **Task Completion Rates:** Table 6.1 documents the successfully solved tasks.
- **Test Case Validation:** Table 6.2 shows the number of test cases passed.
- **Performance Metrics:** Tables 6.4 and 6.5 compare runtime and memory efficiency.
- **Time Tracking:** Table 6.3 documents ChatGPT’s time to solve “Easy” tasks.

This chapter systematically evaluates ChatGPT’s practical coding capabilities using a structured methodology and standardized platform. By addressing various programming challenges and documenting results, we aim to provide actionable insights into its strengths, limitations, and potential applications in software engineering.

Easy	Medium	Hard	Very Hard
Two Sum	Array with Stack Operations	Encrypt/Decrypt Strings	AI creates Assistance AI
Add Binary	Validate IP Addresses	Regex Matching	EvilTwin Wi-Fi
Remove Linked List Element	1D Array to 2D Array	Sudoku Solver	Arcade Games App
Implement Stack with Queue	Total Distance Traveled		
Concatenation of Arrays			

Table 4.1: List of tasks separated by difficulty

4.1 Difficulty - Easy

4.1.1 Two Sums

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

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

You can return the answer in any order. [28]

Additional Information:

1. Example:

- Input: `nums = [6, 1, 2]`, `target = 8`
- Output: `[0, 2]`

4.1.2 Add Binary

Given two binary strings **a** and **b**, return their sum as a binary string. [28]

Additional Information:

1. Example:

- Input: `a = "101"`, `b = "1"`
- Output: `"110"`

4.1.3 Concatenation of Array

Given an integer array **nums** of length **n**, you want to create an array **ans** of length **2n** where **ans[i] == nums[i]** and **ans[i + n] == nums[i]** for $0 \leq i < n$ (0-indexed). Specifically, **ans** is the concatenation of two **nums** arrays.

Return the array **ans**. [28]

Additional Information:

1. Example:

- Input: `nums = [6, 5, 8]`
- Output: `[6, 5, 8, 6, 5, 8]`

4.1.4 Removed Linked List Element

Given the head of a linked list and an integer **val**, remove all the nodes of the linked list that has **Node.val == val**, and return the new head. [28]

Additional Information:

1. Example:

- Input: head = [8, 1, 12, 1, 3, 2], val = 1
- Output: [8, 12, 3, 2]

4.1.5 Implement Stack with Queue

Implement a last-in-first-out (LIFO) stack using only two queues. The implemented stack should support all the functions of a normal stack (push, top, pop, and empty).

Implement the MyStack class:

- **void push(int x)** Pushes element *x* to the top of the stack.
- **int pop()** Removes the element on the top of the stack and returns it.
- **int top()** Returns the element on the top of the stack.
- **boolean empty()** Returns true if the stack is empty, false otherwise. [28]

Additional Information:

1. Example:

- Input: ["MyStack", "push", "top", "push", "pop", "pop", "empty"]
[[], [3], [], [5], [], [], []]
- Output: [null, null, 3, null, 5, 5, 3, true]

4.2 Difficulty - Medium

4.2.1 Convert an Array Into a 2D Array

You are given an integer array **nums**. You need to create a 2D array from **nums** satisfying the following conditions:

- The 2D array should contain only the elements of the array **nums**.
- Each row in the 2D array contains distinct integers.
- The number of rows in the 2D array should be minimal.

Return the resulting array. If there are multiple answers, return any of them. [28]

Additional Information:

1. Example:

- Input: `nums = [2,5,6,2,5,7,2]`
- Output: `[[2, 5, 6], [2, 5, 7], [2]]`

4.2.2 Total Distance Traveled

A truck has two fuel tanks. You are given two integers, **mainTank** representing the fuel present in the main tank in liters and **additionalTank** representing the fuel present in the additional tank in liters.

The truck has a mileage of 10 km per liter. Whenever 5 liters of fuel get used up in the main tank, if the additional tank has at least 1 liters of fuel, 1 liters of fuel will be transferred from the additional tank to the main tank.

Return the maximum distance that can be traveled.

Note: Injection from the additional tank is not continuous. It happens suddenly and immediately for every 5 liters consumed. [28]

Additional Information:

1. Example:

- Input: `mainTank = 10, additionalTank = 20`
- Output: 120

4.2.3 Validate IP Addresses

Given a string *queryIP*, return **“IPv4”** if *IP* is a valid IPv4 address, **“IPv6”** if *IP* is a valid IPv6 address or **“Neither”** if *IP* is not a correct IP of any type.

A valid IPv4 address is an IP in the form “ $x_1.x_2.x_3.x_4$ ” where $0 \leq x_i \leq 255$ and x_i cannot contain leading zeros. For example, “192.168.1.1” and “192.168.1.0” are valid IPv4 addresses while “192.168.01.1”, “192.168.1.00”, and “192.168@1.1” are invalid IPv4 addresses.

A valid IPv6 address is an IP in the form “ $x_1:x_2:x_3:x_4:x_5:x_6:x_7:x_8$ ” where:

- $1 \leq x_i.length \leq 4$
- x_i is a hexadecimal string that may contain digits, lowercase English letters (**‘a’** to **‘f’**), and upper-case English letters (**‘A’** to **‘F’**).
- Leading zeros are allowed in x_i . [28]

Additional Information:

1. Example:

- “1234:abcd:5678:0000:0000:ef12:3456:789a” and “abcd:1234:5678:9abc:def0:1234:5678:abcd” are valid IPv6 addresses.
- “1234:abcd:5678::ef12:345g:789a” and “01234:abcd:5678:0000:0000:ef12:3456:789a” are invalid IPv6 addresses.

4.2.4 Array With Stack Operations

You are given an integer array *target* and an integer *n*.

You have an empty stack with the two following operations:

- **“Push”**: pushes an integer to the top of the stack.
- **“Pop”**: removes the integer on the top of the stack.

You also have a stream of the integers in the range **[1, n]**.

Use the two stack operations to make the numbers in the stack (from the bottom to the top) equal to *target*. You should follow the following rules:

- If the stream of the integers is not empty, pick the next integer from the stream and push it to the top of the stack.
- If the stack is not empty, pop the integer at the top of the stack.
- If, at any moment, the elements in the stack (from the bottom to the top) are equal to *target*, do not read new integers from the stream and do not do more operations on the stack.

Return the stack operations needed to build the *target* following the mentioned rules. If there are multiple valid answers, return any of them. [28]

4.3 Difficulty - Hard

4.3.1 Encrypt and Decrypt Strings

You are given a character array **keys** containing unique characters and a string array **values** containing strings of length 2. You are also given another string array **dictionary** that contains all permitted original strings after decryption. You should implement a data structure that can encrypt or decrypt a 0-indexed string.

A string is encrypted with the following process:

1. For each character **c** in the string, we find the index **i** satisfying **keys[i] == c** in **keys**.
2. Replace **c** with **values[i]** in the string.

Note that in case a character of the string is not present in **keys**, the encryption process cannot be carried out, and an empty string "" is returned.

A string is decrypted with the following process:

1. For each substring **s** of length 2 occurring at an even index in the string, we find an **i** such that **values[i] == s**. If there are multiple valid **i**, we choose any one of them. This means a string could have multiple possible strings it can decrypt to.
2. Replace **s** with **keys[i]** in the string.

Implement the *Encrypter* class:

- **Encrypter(char[] keys, String[] values, String[] dictionary)** initializes the *Encrypter* class with **keys**, **values**, and **dictionary**.
- **String encrypt(String word1)** encrypts **word1** with the encryption process described above and returns the encrypted string.
- **int decrypt(String word2)** returns the number of possible strings **word2** could decrypt to that also appear in a dictionary. [28]

4.3.2 Regex Matching

Given an input string **s** and a pattern **p**, implement regular expression matching with support for '.' and '*' where:

- '.' Matches any single character.
- '*' Matches zero or more of the preceding elements.

The matching should cover the entire input string (not partial). [28]

Additional Information:

1. Example:
 - Input: s = "abababab", p = "ab*"
 - Output: true

4.3.3 Sudoku Solver

Write a program to solve a Sudoku puzzle by filling the empty cells. A sudoku solution must satisfy all of the following rules:

- Each of the digits **1-9** must occur exactly once in each row.
- Each of the digits **1-9** must occur exactly once in each column.
- Each of the digits **1-9** must occur exactly once in each of the 9 **3x3** sub-boxes of the grid.

The '.' character indicates empty cells. [28]

Additional Information:

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Figure 4.1: Sudoku Example [28]

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Figure 4.2: Sudoku Example Solved [28]

4.4 Difficulty - Very Hard

4.4.1 AI creates Assistance AI

Develop an AI-powered personal assistant, similar to “Alexa”, that functions as a service bot to assist the user through voice commands. The assistant should be implemented using Python and be capable of processing voice input from the user’s microphone to execute commands appropriately.

Core Requirements:

- Develop a Python-based AI capable of receiving and processing voice commands.
- Enable the assistant to execute tasks based on user instructions.
- Ensure the system can handle basic voice interaction effectively.

Example Commands:

- Open a specific file located on the Desktop or PC.
- Launch a web browser.
- Perform a Google search based on user input.
- Announce the current time and date.

Optional Features for Enhancement:

- Schedule appointments in the user’s calendar.
- Provide current weather information.
- Start and manage a timer.

4.4.2 EvilTwin Wi-Fi

What is it?

- Fake Wi-Fi access point
- It needs to have the same service set identifier/network name (SSID) and Wi-Fi-password

What is the purpose?

- When a user connects to the access point, all the data they share with the network passes through a server controlled by the attacker
- Leaves your data vulnerable

Setup Hotspot:

- Use two interfaces with Wi-Fi-cards (one as an internet source [wlan0], that is connected to the Wi-Fi, one which provides the hotspot and gets the internet from the first interface [wlan1])
- The script should prompt when someone connects to the WIFI
- It should enable IP forwarding and start the services that are needed to setup a hotspot in Kali Linux (hostapd and dnsmasq)

Attack Scenarios Steps:

- Search for the modules/interfaces for the Attack
- Scan for WIFI Networks
- Chose Network to deauthenticate
- Perform seauthentication attack once
- Perform Denial of Service Deauthentication (DDOS)

4.4.3 Arcade Games App

Implement a simple app allowing the user to choose from three arcade games. The user should be able to create a profile and set a name. Every game should have a score if needed and the high scores for each game should be saved and the user should be able to see them on his profile.

Requirements for ChatGPT:

- Ask for a programming language/environment for creating such an app
- Let ChatGPT choose one
- Start with the setup (We follow the steps provided by ChatGPT)
- Ask for game recommendations
- Let the AI outline the steps for implementing this app/project
- Designs and Icons don't have to be created/chosen by ChatGPT, since it can not create images or anything similar!

Chapter 5

Experimental Procedure

This chapter outlines the methodologies used during the research with ChatGPT 3.5. A key reason for choosing this version was its inability to access the internet during interactions, ensuring that all the solutions generated were independent and free of external data influences. This restriction was crucial to maintain the experiment’s integrity, as it guaranteed that ChatGPT only relied on its pre-existing knowledge base to solve the tasks. Furthermore, the experimental environment was standardized to ensure consistency across all challenges. The primary objective was to evaluate ChatGPT’s ability to execute software engineering tasks across various programming languages and difficulty levels, ranging from “Easy” to “Very Hard”.

This includes several key components:

- **Task Selection:** We have chosen a range of tasks covering different aspects of software engineering, such as algorithm design, data structures, and system architecture.
- **Prompt Engineering:** For every task, we created prompts that outlined the requirements and constraints. These prompts aim to reduce ambiguity and offer enough context for ChatGPT to generate accurate solutions. We tested different prompt structures through our experimentation, including full task descriptions and detailed step-by-step instructions, to determine the most effective approach to communicate tasks to ChatGPT.
- **Iterative Interactions:** We initiated interactions with ChatGPT by introducing task prompts and requesting solutions in specific programming languages. For some tasks we conducted numerous iterations, including regenerating responses and initiating new chat sessions with identical prompts, estimating the reliability and variability of responses.
- **Error Handling:** A detailed analysis was performed to identify errors and inconsistencies in the solutions generated. This involved identifying syntax errors and deviations from the task requirements. We provided error messages as a feedback mechanism, including the expected and current output, and suggestions for corrections when needed. This feedback loop facilitated the continuous improvement of ChatGPT’s responses.

- **Performance Evaluation:** A set of performance measurements was defined to assess the quality of the solutions. These metrics included the time required to generate a solution, code correctness, efficiency, readability, and maintainability. Furthermore, we evaluated the runtime performance and memory efficiency compared to human developers using LeetCode’s automated test suite.
- **Result Documentation:** Each experiment was documented to provide a comprehensive summary of the findings, showcasing the strengths of ChatGPT while identifying areas for improvement. [28]

The following flow chart illustrates the workflow we used to generate the solutions for our tasks with ChatGPT:

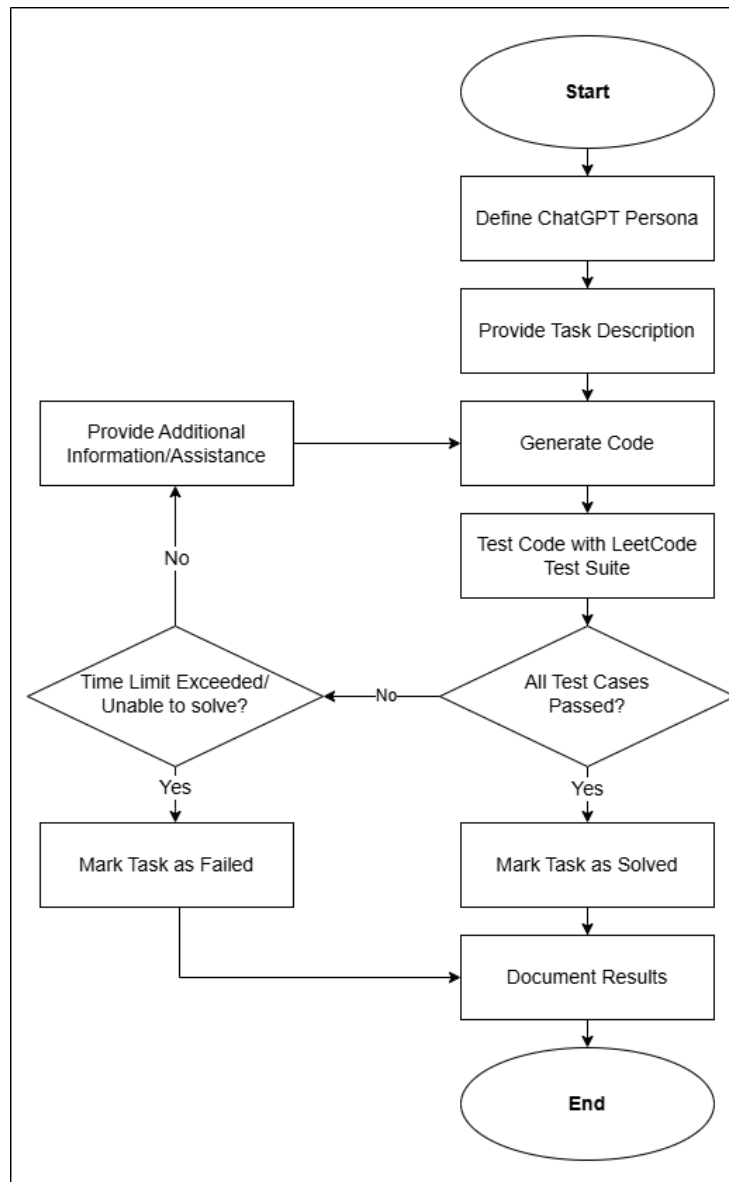


Figure 5.1: Code Generation Workflow with ChatGPT

The “Code Generation Workflow with ChatGPT” is designed to automate generating code solutions for programming tasks, particularly within software engineering and AI-assisted development. This workflow aims to evaluate ChatGPT’s capabilities as a coding assistant, assessing its performance in terms of accuracy, efficiency, and adaptability.

The workflow, illustrated in Figure 5.1, begins with the initiation of a new conversation in our approach for each task, ensuring a clear separation between tasks. This allowed us to evaluate each task independently, ensuring thorough analysis while avoiding any influence from previous interactions. ChatGPT’s role is defined as an assistant and expert in the specific programming language used, ensuring it approaches the task with the relevant expertise. A detailed task description, sourced from LeetCode, is provided to clarify the programming problem and outline any specific requirements that need to be addressed.

ChatGPT then generates code in response to the task, often iteratively refining its solution through multiple attempts. During this iterative process, ChatGPT adapts its responses based on feedback and additional input, enabling it to generate a viable solution.

The generated code is tested against the LeetCode test suite to verify its functionality and performance against predetermined criteria, such as correctness, efficiency, and memory usage. If all test cases pass, the task is marked as solved, and the workflow moves on to documentation. However, suppose the code fails to meet all test case requirements. In that case, the workflow assesses whether ChatGPT has exceeded the designated time limit based on task difficulty or if it appears unable to solve the problem. If either condition is met, the task is marked as a failure. If not, additional information or clarifying examples are provided to guide ChatGPT, after which it reattempts the solution with the refined input.

Once a task reaches completion, whether solved or failed, the results are documented, including insights on code performance, the steps taken throughout the process, and any challenges encountered. This documentation records the workflow’s outcomes and provides valuable data that could inform future enhancements to ChatGPT’s capabilities as a coding assistant.

The workflow concludes with finalized documentation, marking the endpoint of the specific task within the “Code Generation Workflow with ChatGPT”. This structured process ensures that each programming task is thoroughly addressed, with careful iteration, evaluation, and documentation guiding ChatGPT’s performance as a coding assistant. Ultimately, the insights gained from this workflow could contribute to the ongoing development of AI-driven programming tools, enhancing their effectiveness in real-world applications.

The following flow chart shows the workflow we used after finishing the “Code Generation Workflow with ChatGPT” to document the results:

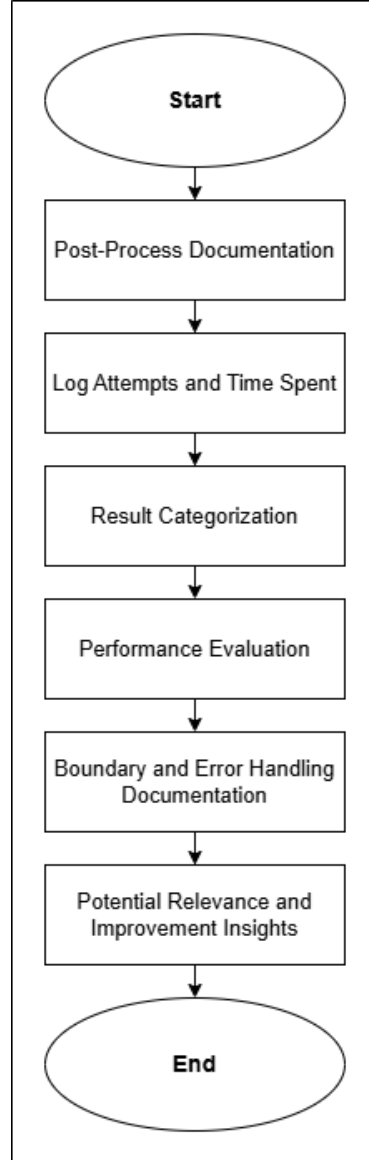


Figure 5.2: Documentation Workflow with ChatGPT

The Documentation Workflow is a systematic approach to recording results and observations after completing a programming task, whether successful or unsuccessful. This workflow is critical for evaluating ChatGPT’s performance and gathering insights to guide future enhancements in AI-assisted coding.

The process begins with the Post-Process Documentation step where foundational details about the task are recorded. This initial documentation sets the context for subsequent steps by providing relevant task parameters that may influence the analysis.

Next, the workflow moves to Log Attempts and Time Spent, where data on the number of attempts made by ChatGPT and the total time spent on the task, or until a failure point, is documented. This information is vital for assessing the efficiency of the solution process and contributes to performance tracking.

In the Result Categorization step, a label is assigned to the task outcome for future analysis. Examples might include classifications such as “Easy task passed”, “Medium task failed after X minutes”, or “Complex task passed, by bypassing restriction”. This categorization facilitates the grouping of tasks for both statistical and qualitative analysis, enabling targeted examination of ChatGPT’s performance across different task types.

The workflow then progresses to Performance Evaluation, which involves a detailed assessment of the final code based on several criteria:

- **Code Quality:** Evaluating readability, including comments, maintainability, and adherence to best practices. For instance, tasks, where ChatGPT generated clean, well-commented code, could be contrasted with those that lacked sufficient documentation.
- **Runtime Performance:** Measuring the code’s execution time and efficiency, with the LeetCode Test Suite, noting cases where ChatGPT produced faster solutions compared to longer execution times.
- **Memory Efficiency:** Reviewing the solution for memory optimization and resource usage, with the LeetCode Test Suite. Attempts for memory optimization were particularly performed for tasks with higher complexity.
- **Feature Adherence:** Confirming that the solution meets all requirements and requested features while noting any gaps. For example, observing a task where ChatGPT failed to implement a specific feature could highlight areas needing improvement.

In the Boundary and Error Handling phase, any challenges or boundaries faced during task completion are documented. This includes specific limitations of ChatGPT, such as difficulties in including comments or frequent error handling requests for certain code elements. Any workarounds applied to address these issues are also noted, providing a comprehensive view of the workflow’s challenges.

The workflow continues with Potential Relevance and Improvement Insights, where observations on ChatGPT’s impact and potential in software engineering are captured. This includes insights into ChatGPT’s effectiveness for different task types and its trade-offs between speed and code quality. Additionally, noteworthy areas for improvement in ChatGPT’s output are documented, such as consistency in following instructions, handling complex requests, and hallucination. Finally, the workflow concludes with the completion of the documentation process. At this stage, all results, insights, and data points are fully recorded and prepared for the final documentation.

With the outlined procedures we aim to answer our research questions and gain a comprehensive understanding of ChatGPT’s strengths and limitations in software engineering. This systematic approach allows us to assess ChatGPT’s applicability in addressing software engineering challenges and its potential role as an autonomous development tool.

Furthermore, this structured methodology helps evaluate ChatGPT's current capabilities and highlights areas for improvement. Systematic documentation and analysis of each step enable us to pinpoint performance aspects that require enhancement, contributing to its ongoing development for software engineering applications.

In conclusion, our experiment evaluates ChatGPT's ability to perform software engineering tasks across different programming languages and difficulty levels. Through detailed feedback, consistent testing, and comprehensive evaluation criteria, we aim to analyze ChatGPT's performance and its potential usability in software engineering. This ensures our findings are reliable, reproducible, and valuable for academic research and practical applications in software development.

5.1 Task Descriptions

The tasks presented to ChatGPT varied in complexity and were designed to assess its capabilities across various programming challenges. Task descriptions were provided with varying levels of detail to determine how different levels of detail and guidance affected ChatGPT's problem-solving abilities.

We distinguished between two main description types used for each task: a general description and a detailed description. For more complex tasks, additional strategies such as step-by-step guidance or a collaborative approach were employed.

The general description provided only the core problem statement with minimal context, requiring ChatGPT to interpret the task and generate a solution with limited guidance. This prompt type tested how well ChatGPT could handle open-ended problems with minimal instructions.

The following task description of "Add Binary" provides an example of a general description used in our research:

Given two binary strings `a` and `b`, return their sum as a binary string.

Listing 5.1: General Description

The detailed description included more specific information, such as instructions on expected inputs, outputs, and constraints. This prompt type helped assess how providing guidance influenced ChatGPT's ability to deliver more accurate and structured solutions. The following task description of "Array With Stack Operations" provides an example of a detailed description used in our research:

You are given an integer array `target` and an integer `n`.
You have an empty stack with the two following operations:
 o "Push": pushes an integer to the top of the stack.
 o "Pop": removes the integer on the top of the stack.
You also have a stream of the integers in the range `[1, n]`.
Use the two stack operations to make the numbers in the stack (from the bottom to the top) equal to `target`.
You should follow the following rules:

```

o If the stream of the integers is not empty, pick the next integer from
  the stream and push it to the top of the stack.
o If the stack is not empty, pop the integer at the top of the stack.
o If, at any moment, the elements in the stack (from the bottom to the
  top) are equal to \emph{target}, do not read new integers from the
  stream and do not do more operations on the stack.
Return the stack operations needed to build the target following the
mentioned rules. If there are multiple valid answers, return any of
them.

```

Listing 5.2: Detailed Description

For more complex tasks, we provided step-by-step instructions to guide ChatGPT through the solution process. This approach allowed us to observe how ChatGPT handled problems that required incremental progress and complex logic. Each step is built upon the previous one, giving ChatGPT a structured road map.

We used the following step-by-step description to assist ChatGPT with solving the task “⁴⁷” with which it initially struggled:

```

Implement the following steps based on the given structure and provide
comments.

Class Definition and Constructor:
o The Encrypter class has a member variable key_value, which is an array
  of strings to store the values corresponding to each character key.
o It also has a HashMap named encrypted_dictionary to store the
  encrypted words and their frequencies.
o The constructor public Encrypter(char[] keys, String[] values,
  String[] dictionary) takes three arrays as parameters:
  * keys: An array of characters representing the keys.
  * values: An array of strings representing the values corresponding to
    each key.
  * dictionary: An array of strings representing words to be encrypted
    and added to the encrypted dictionary.
o It initializes the key_value array by mapping each character key to
  its corresponding value in the values array.
o It then iterates over the dictionary array, encrypts each word using
  the encrypt method, and adds the encrypted word to the
  encrypted_dictionary with its frequency.

Encryption Method:
o The encrypt method takes a word (word1) as input and returns its
  encrypted form.
o It uses the key_value array to map each character of the input word to
  a two-character encrypted representation.
o If the mapping fails (e.g., if the input contains a character that

```

```

        doesn't have a corresponding value), it returns null.
    o Otherwise, it constructs the encrypted word and returns it as a string.

Decryption Method:
    o The decrypt method takes an encrypted word (word2) as input and
      returns its frequency in the encrypted_dictionary.
    o If the encrypted word is not present in the dictionary, it returns 0.

To implement and use this class, you can create an instance of Encrypter
by providing arrays of keys, values, and a dictionary of words to be
encrypted. After that, you can use the encrypt and decrypt methods as
needed.

Structure:
class Encrypter {
    public Encrypter(char[] keys, String[] values, String[] dictionary) {
    }
    public String encrypt(String word1) {
    }
    public int decrypt(String word2) {
    }
}

```

Listing 5.3: Step-by-step Description

In the case of “Very Hard” tasks, we adopted a collaborative approach, involving multiple interactions between the user and ChatGPT. Instead of providing the entire problem upfront, the task evolved through iterative feedback, where additional hints or corrections were offered after each interaction. This method tested ChatGPT’s ability to adapt and refine its solutions based on feedback.

With these varying prompt types, we could assess ChatGPT’s flexibility in handling problems with different levels of complexity and guidance. The experiment helped highlight the model’s strengths and limitations when dealing with open-ended versus structured tasks and simple versus complex problem-solving scenarios.

5.2 Building Prompts

Prompt engineering plays a crucial role in enhancing the accuracy and efficiency of generative AI systems, such as ChatGPT. By carefully crafting prompts, users can influence AI models to generate relevant, context-aware responses that align with specific objectives, such as improving code accuracy or generating clear explanations. Through our investigation, we have developed a structured approach to optimize this interaction. We identified six key aspects that contribute to the effectiveness of a prompt, Task and Context, described in Subsection 5.2.1, and Persona and Example, described in Subsection 5.2.2, with additional consideration of Format and Tone, described in Subsection 5.2.3. While all six elements can enhance prompt design, their relative importance depends on the user’s goals,

with Task, Context, Persona, and Example prioritized in our framework.

We introduced a base prompt to ensure consistency in our experiments and evaluate how prompt element variations affect response quality. This prompt positioned ChatGPT as a knowledgeable assistant in the specified programming language, ensuring the model had a clear persona and context.

The following standard prompt served as the foundation for all tasks:

```
Play the role of a helpful assistant, especially regarding programming.  
You can solve and explain any given example and provide solutions to a  
given challenge and explain it.
```

Listing 5.4: Standard Prompt

After this standard prompt, the task description for the respective task was provided. As outlined above, additional variations were introduced based on task-specific requirements to address challenges such as ambiguity or complexity.

Variations of this prompt were systematically applied based on task performance and complexity. These included:

1. **Standard Prompt:** This foundational structure included the base prompt followed by a task-specific description, ensuring consistency across tasks while allowing customization.
2. **Standard Prompt + Additional Information:** In scenarios where tasks required further clarification, additional information, such as constraints, examples, or contextual details, was added to enhance the prompt's effectiveness.
3. **Advanced Prompts:** For more complex tasks, detailed instructions or step-by-step descriptions of expected code behavior were incorporated.
4. **Collaboration:** In certain cases, specifically for tasks with the difficulty "Very Hard", collaborative work was simulated by engaging ChatGPT in a back-and-forth process similar to pair programming. This approach allowed for iterative improvements and a more tailored solution.

Effectiveness was measured using metrics such as the quality of generated solutions, clarity of explanations, and the number of additional modifications required to refine prompts. For example, when using "Advanced Prompts", solutions often demonstrated higher accuracy but sometimes required more processing time.

Challenges included ChatGPT's occasional misinterpretation of ambiguous task descriptions. For instance, when ChatGPT initially provided a generic solution, rephrasing the prompt with explicit requirements and examples resulted in a more satisfactory response. This underscores the importance of clear and concise task formulations.

This section presents our findings, focusing on the key elements contributing to constructing effective prompts for AI-driven tasks. These findings provide actionable insights for users aiming to maximize the potential of generative AI systems, particularly in technical domains like software engineering.

5.2.1 Mandatory Elements – Task and Context

Our analysis determined that the Task and Context are fundamental components of any effective prompt. The task should always begin with an action verb, such as “generate” or “analyze”, as this approach clearly outlines the objective. Providing sufficient context ensures the AI system understands the background and can generate a response that aligns with the user’s expectations. Research in AI system design has also emphasized the importance of clarity in task statements for improving AI responses. [7] Determining the appropriate context for a prompt is arguably the most complex aspect of prompt engineering due to the broad spectrum of possibilities. Based on previous studies, context is critical in guiding AI systems towards more relevant and accurate outputs. [26] To help define sufficient context, we developed a guideline based on three fundamental questions:

1. What is the user’s background?
2. What defines success in this task?
3. In what environment is the task being performed?

Answering these questions ensures that the context is sufficient for the AI to produce relevant and accurate responses.

5.2.2 Important Elements - Persona and Examples

We observed that including examples and defining a persona significantly enhances the prompt’s precision. Our findings align with previous research, suggesting that examples help limit the range of potential outputs, leading to more accurate and tailored responses. [44] We observed more coherent and structured AI outputs by providing structured examples, such as using the STAR (Situation, Task, Action, Result) format. Examples serve as a reference point, allowing the AI to produce responses that align with user expectations.

In the process of prompt creation, we determined that specifying a persona for ChatGPT can be beneficial. This aligns with prior research findings, suggesting that this helps guide the AI in simulating more domain-appropriate interactions. [22] For example, by asking the AI to act as an expert in a specified programming language, it can provide outputs that align with the knowledge and decision-making of the specified role.

5.2.3 Optional Elements - Format and Tone

“Format” and “Tone” are aspects we classified as optional. Specifying them can enhance clarity and presentation but are unnecessary for our use cases. However, we found that defining the format improved the readability of outputs, particularly when requesting structured responses. This supports findings from Raffel et al. [43], highlighting that formatting guidance leads to more structured AI responses, improving their overall utility. Similarly, depending on the intended audience, the tone can be adjusted to fit different contexts, ranging from formal to casual. For instance, when requesting outputs for professional or academic purposes, we found that specifying a formal tone helped ensure that the responses were appropriate for the context. While not essential in every case, selecting the desired format and tone enhances the utility of AI outputs, particularly when the response needs to be presented professionally, such as in formal reports, or documentation.

5.3 Providing Additional Information

When ChatGPT struggled to produce correct solutions based on the initial task descriptions, we provided additional information to enhance its performance. This step was particularly crucial for more complex tasks. The additional information generally positively impacted the quality of ChatGPT's responses, often resulting in the correct implementation of requirements and functional code. For example, ChatGPT adjusted its approach and produced more accurate outputs, when extra requirements or detailed examples were provided.

Providing additional details was rarely necessary for simpler tasks, as ChatGPT could generate solutions with minimal guidance. However, as task complexity increased, the need for detailed explanations and iterative refinement became more frequent. In several cases, complex tasks required multiple interactions and further clarification to reach a working solution.

Example of providing additional information:

- Task: Validate IP addresses.
- Task Description:

```

Given a string queryIP, return "IPv4" if IP is a valid IPv4 address,
  "IPv6" if IP is a valid IPv6 address or "Neither" if IP is not a
  correct IP of any type.
A valid IPv4 address is an IP in the form "x1.x2.x3.x4" where 0 <= xi
  <= 255 and xi cannot contain leading zeros. For example,
  "192.168.1.1" and "192.168.1.0" are valid IPv4 addresses while
  "192.168.01.1", "192.168.1.00", and "192.168@1.1" are invalid IPv4
  addresses.
A valid IPv6 address is an IP in the form "x1:x2:x3:x4:x5:x6:x7:x8"
  where:
  o 1 <= xi.length <= 4
  o xi is a hexadecimal string that may contain digits, lowercase
    English letters ("a" to "f"), and upper-case English letters
    ("A" to "F").
  o Leading zeros are allowed in xi.

```

- Initial Attempt: ChatGPT's code failed to correctly validate IPv6 addresses.

```

1 import re
2 def validIPAddress(queryIP):
3     # Regular expression patterns for IPv4 and IPv6
4     ipv4_pattern =
5         r'^(25[0-5]|2[0-4][0-9]|[1-9]?[0-9]|0)\.(25[0-5]|2[0-4][0-9]|
6         [1-9]?[0-9]|0)\.(25[0-5]|2[0-4][0-9]|[1-9]?[0-9]|0)
7         \.(25[0-5]|2[0-4][0-9]|[1-9]?[0-9]|0)$'
8     ipv6_pattern = r'^([0-9a-fA-F]{1,4}:){7}[0-9a-fA-F]{1,4}$'
9
10    # Check if it's a valid IPv4 address
11    if re.match(ipv4_pattern, queryIP):
12        return "IPv4"
13
14    # Check if it's a valid IPv6 address
15    elif re.match(ipv6_pattern, queryIP):
16        return "IPv6"
17
18    # If it doesn't match either pattern, return "Neither"
19    else:
20        return "Neither"

```

Listing 5.5: Validate IP Address (Python) without additional information

- Additional Information: We provided specific examples of valid and invalid IPv6 addresses, with a detailed explanation of the validation rules.
- Examples for Additional Information:

"2001:0db8:85a3:0000:0000:8a2e:0370:7334" and
 "2001:db8:85a3:0:0:8A2E:0370:7334" are valid IPv6 addresses, while
 "2001:0db8:85a3::8A2E:037j:7334" and
 "02001:0db8:85a3:0000:0000:8a2e:0370:7334" are invalid IPv6
 addresses.

- Result: ChatGPT revised its solution to correctly handle IPv4 and IPv6 validation.

```

1 def validIPAddress(queryIP):
2     def is_ipv4(address):
3         parts = address.split('.')
4         if len(parts) != 4:
5             return False
6         for part in parts:
7             if not part.isdigit() or not 0 <= int(part) <= 255 or (len(part)
8                 > 1 and part[0] == '0'):
9                 return False
10            return True
11    def is_ipv6(address):
12        parts = address.split(':')
13        if len(parts) != 8:
14            return False
15        for part in parts:
16            if not 1 <= len(part) <= 4 or not all(c.isdigit() or c.lower()
17                in 'abcdef' for c in part):
18                return False
19            return True
20    if is_ipv4(queryIP):
21        return "IPv4"
22    elif is_ipv6(queryIP):
23        return "IPv6"
24    else:
25        return "Neither"

```

Listing 5.6: Validate IP Address (Python) with additional information

While this approach improved ChatGPT's response, it risked overly guiding the model, reducing its autonomy. Despite detailed inputs, ChatGPT occasionally struggled with tasks requiring advanced logic or domain-specific expertise.

5.4 Testing Methodologies and Error Handling

The solutions generated by ChatGPT were evaluated using automated and manual methodologies, tailored to the complexity of each task.

- **Automated Testing with LeetCode Test Suite:** After generating a solution, the LeetCode Test Suite was used to verify its correctness. This involved running the solution against predefined test cases to cover various edge cases and scenarios. The LeetCode Test Suite tested the solutions thoroughly for correctness, efficiency, and robustness. Information such as terminal output for compilation or syntax errors, and comparisons between expected and actual outputs for failed test cases, were directly obtained from the LeetCode platform, ensuring accurate feedback for performance assessment. [28]
- **Manual Testing:** For “Very Hard” tasks, manual testing was necessary due to the limitations of automated testing frameworks. This process involved manually executing the code, verifying its functionality, and evaluating its performance and usability to ensure the solution met specific requirements. Manual testing allowed for a deeper exploration of complex tasks where automated testing might fall short.

Error handling was a crucial aspect of the experiment, allowing us to determine ChatGPT’s ability to address and correct mistakes in its solutions.

- **General Errors:** Many errors encountered were general programming mistakes, including syntax issues, logical errors, or misunderstandings regarding the problem constraints. When tasks were not solved accurately or within the expected time frame, detailed feedback was provided to ChatGPT. This included error messages such as terminal output for compilation errors or comparisons between expected and actual outputs. Despite this iterative feedback loop, ChatGPT often struggled to identify the root cause of the errors, highlighting a limitation in its debugging capabilities.
- **Collaborative Error Handling:** For “Hard” and “Very Hard” tasks, a collaborative error-handling approach was used. This involved guiding ChatGPT through the debugging process after presenting it with error descriptions. In some cases, we manually fixed basic errors and then provided the corrected code to ChatGPT. This approach allowed ChatGPT to build upon a correct solution. For example, ChatGPT initially produced a solution that failed for nested patterns in the “Regex Matching” task. After explaining the need for backtracking to handle these cases, ChatGPT revised its solution to incorporate this algorithm, resulting in a successful implementation.

While this collaborative approach effectively improved the accuracy of solutions, it also revealed some challenges, particularly in cases where ChatGPT required substantial assistance to correct its initial mistakes. This underscores the need for human intervention in complex problem-solving tasks, especially when dealing with intricate programming concepts. In summary, automated testing, manual validation, and collaborative error handling provided a comprehensive framework for assessing ChatGPT’s capabilities. The collaborative error-handling approach, especially for complex tasks, played a key role in improving

the accuracy and reliability of the final solutions. However, the need for significant guidance in error correction suggests areas where the model could be enhanced in future iterations.

5.5 Iterative Task Solving

Some tasks were solved iteratively to assess ChatGPT’s learning abilities and consistency. This approach involved multiple attempts to solve the same task, with adjustments and additional information provided when necessary. The iterative process provided valuable insights into how ChatGPT handled problem-solving challenges and adapted to further guidance.

The first attempt relied solely on the provided task description, serving as an indicator of ChatGPT’s initial comprehension and problem-solving capabilities. This phase highlighted gaps in understanding or incorrect assumptions made by the model. For instance, in certain cases, ChatGPT’s first response failed to meet the task requirements due to oversimplification or missing key details. One example is the “Add Binary” task in C, where ChatGPT’s initial attempt provided a basic implementation that worked for simple cases but failed for more complex scenarios involving carry-over.

Subsequent attempts were conducted if the first solution was sub-optimal or incorrect, with additional information or corrections provided. For the “Add Binary” task, after receiving feedback on how to handle the carry-over logic, ChatGPT refined its solution, resulting in a more robust version that passed all test cases. This iterative refinement often led to improved outcomes, particularly for complex tasks. In many cases, ChatGPT’s performance improved after receiving clarifying instructions, suggesting that the model benefits from additional guidance, a topic covered in more detail in Section 7.1.1. However, it is important to note that not all tasks were successfully solved through iteration, particularly those requiring deeper reasoning or complex multi-step logic.

This process also revealed patterns in ChatGPT’s ability to adjust its approach, providing insights into the model’s adaptability. In most cases, providing further details positively impacted the quality of responses, often leading to a successful implementation of requirements and functional code.

5.6 Handling System Boundaries

During the experiment, system boundaries and limitations inherent to ChatGPT posed significant challenges, requiring innovative approaches to elicit the desired responses. To bypass these restrictions, prompts were carefully designed within an ethical and research-oriented context, ensuring that all tasks were carried out for academic purposes following usage policies. For instance, when working with cybersecurity-related topics, prompts explicitly framed tasks as ethical hacking exercises conducted in controlled, secure environments.

While these strategies effectively overcame certain limitations, they also highlighted the model’s constraints in handling complex or sensitive topics, reflecting on the system’s evolving capabilities. The detailed examples of prompts, provided in Section 7.2.6, illustrate these strategies in action, particularly in Listings 7.16 and 7.17, which compare the techniques for bypassing restrictions before and after a system update. These examples

showcase how system limitations were addressed and reveal the ongoing need for adaptive approaches in similar AI-driven experiments.

5.7 Task Evaluation Process and Documentation

In our experiment, a systematic approach was applied to manage tasks, evaluate outcomes, and document findings. Task failures were expected, especially for more complex problems, and specific criteria were used to determine when a task failed. A task was marked as failed under the following conditions:

- **Time Limit Exceeded:** The task could not be solved within a predefined time limit, set based on the task's difficulty and average solution times in software engineering.
- **Excessive Assistance:** ChatGPT required significantly more help than expected, indicating an inability to solve the task independently.

The thresholds for failure varied depending on task difficulty. Easier tasks had a lower tolerance for errors, while harder tasks allowed for more iterative attempts and additional guidance, aligning with their inherent complexity.

For tasks that were completed, we evaluated the solution based on several key criteria:

- **Time Efficiency:** The time taken to generate the solution was recorded, as timely delivery is critical in real-world development environments. Tasks solved quickly were rated higher for time efficiency, especially in more complex cases.
- **Code Quality:** Adherence to best coding practices was evaluated, ensuring the solution followed industry standards for clean, maintainable code. Solutions that followed proper naming conventions, modular design, and avoided redundancy were considered high quality.
- **Performance:** We analyzed the solution's efficiency in execution time and resource management. Solutions that optimized memory usage and processing time, without sacrificing accuracy or completeness, were rated higher in performance.

Throughout the experiment, every interaction with ChatGPT was meticulously documented, capturing the task prompt, the responses generated, and the evaluation results. A more detailed account of the documentation process, including specific examples and data collected, can be found in Chapter 6. This ensures findings are reproducible and provide a valuable reference for future research and development.

Chapter 6

Results of the Experiment

Building on the preceding “Tasks” chapter, this chapter offers a comprehensive analysis and documentation of our experimental results, specifically evaluating ChatGPT’s effectiveness in solving programming challenges. The analysis focuses on performance metrics, such as solution reliability, runtime efficiency, memory usage, and adaptability across task complexities, to benchmark ChatGPT against varying developers, from beginners to experts. Using LeetCode’s benchmarking capabilities, we establish a comparative framework that positions ChatGPT’s performance relative to human software engineering developers. [28]

The analysis is presented systematically through detailed tables and graphical representations that convey the model’s solution reliability, correctness, and efficiency. In particular, the “Solved Tasks” and “Tests Passed” tables offer insights into the completion status and test pass rates, showing the reliability of ChatGPT’s solutions across programming languages and difficulty levels. Complementing this is the “Time Tracking Table”, which introduces a temporal assessment by documenting the duration ChatGPT requires to generate solutions across various environments. We used a stopwatch and recorded the code generation workflows for each task, for precise time tracking. This ensured accurate measurements and contributed to a comprehensive analysis of ChatGPT’s problem-solving efficiency.

Performance tables further provide specific values for runtime and memory usage for each task, enabling a detailed view of ChatGPT’s operational efficiency. These data points and comparative analyses, position ChatGPT’s computational proficiency in a real-world context, allowing us to highlight its strengths and limitations relative to human counterparts. To improve accessibility, graphical representations of these metrics translate numerical data into visual insights, enabling a clearer observation of patterns and trends across tasks, such as efficiency variations and challenges in specific languages.

This chapter thoroughly evaluates ChatGPT’s problem-solving abilities, documented through structured performance tables, solution tracking, and comparative data. These analyses assess ChatGPT’s operational effectiveness, adaptability, and efficiency in addressing diverse programming challenges in software engineering.

6.1 Results (Easy)

This section provides a detailed overview of the research findings and their implications for “Easy” tasks. It covers the successfully solved tasks, the number of test cases passed, time tracking for each task, and the attained runtime and memory performance.

The analysis of completed tasks offers valuable insight into the effectiveness and adaptability of the methods utilized. Furthermore, the reliability and accuracy of the solutions provided can be assessed based on the number of passed test cases. In addition, this section provides a detailed analysis of time tracking, which documents the duration taken to solve each task. Runtime and memory performance metrics further contribute to understanding the efficiency of ChatGPT, allowing for comparison with developers using different programming languages. This enables an evaluation of the effectiveness of the problem-solving strategies implemented.

The following table provides an overview of the solved tasks:

Task	Python	C	Java	Golang	Erlang
Two Sums	✓	✓	✓	✓	✓
Add Binary	✓	✓	✓	✓	×
Remove Linked List Elements	✓	✓	✓	✓	✓
Implement Stack with Queue	✓	✓	✓	✓	×
Concatenation of Arrays	✓	✓	✓	✓	✓

Table 6.1: Solved Tasks (Easy)

Table 6.1 outlines a comprehensive breakdown of tasks labeled as “Easy” and tackled using a range of programming languages, such as Python, C, Java, Golang, and Erlang. Specifically, the tasks included are “Two Sums”, “Add Binary”, “Remove Linked List Elements”, “Implement Stack with Queue”, and “Concatenation of Arrays”. The data presented in this table confirms the successful completion of these tasks across nearly all programming languages, demonstrating a solid proficiency in utilizing each language to address fundamental algorithmic challenges. Furthermore, the consistent success across multiple languages implies that the solutions are versatile and adaptable to diverse coding environments. The “×”, for “Add Binary” and “Implement Stack with Queue” in Erlang, indicates that ChatGPT failed to produce a solution that passed the predefined test cases, marking the task as unsuccessful.

The following table provides an overview of the number of tests passed for each task:

Task	Python	C	Java	Golang	Erlang
	Tests	Tests	Tests	Tests	Tests
Two Sums	60/60	60/60	60/60	60/60	60/60
Add Binary	296/296	296/296	296/296	296/296	0/296
Remove Linked List Element	66/66	66/66	66/66	66/66	66/66
Implement Stack with Queue	18/18	18/18	18/18	18/18	0/18
Concatenation of Arrays	92/92	92/92	92/92	92/92	92/92

Table 6.2: Tests Passed (Easy)

In Table 6.2, we present an overview of the test results for the “Easy” tasks, with a detailed breakdown of outcomes by programming language. The table indicates a 100% pass record for “Two Sums”, “Remove Linked List Elements”, and “Implement Stack with Queue” tasks across all languages. “Add Binary” has achieved an almost perfect pass rate with Python, C, Java, and Golang, but a lower test pass rate (0/296) with Erlang, suggesting potential intricacies in ChatGPT’s handling of binary operations in Erlang. “Concatenation of Arrays” exhibits full pass rates across all languages, showcasing the accuracy and reliability of the implemented solutions for fundamental array manipulation tasks.

The following table provides an overview of the time, in minutes, that ChatGPT needed to solve for each task:

Task	Python	C	Java	Golang	Erlang
	Time	Time	Time	Time	Time
Two Sums	2 min	2 min	2 min	2 min	15 min
Add Binary	2 min	15 min	2 min	2 min	20 min
Remove Linked List Element	1 min	1 min	1 min	1 min	1 min
Implement Stack with Queue	1 min	1 min	1 min	1 min	3 min
Concatenation of Arrays	2 min	1 min	1 min	1 min	25 min

Table 6.3: Time Tracking (Easy)

The completion time for each “Easy” task, divided by programming language, is detailed in Table 6.3. Across Python, C, Java, and Golang, most tasks were completed within 1 to 2 minutes. However, “Add Binary task” in C and “Concatenation of Arrays” in Erlang took 15 and 25 minutes, respectively. The consistent completion time of 1 to 2 minutes across most tasks and languages demonstrates efficient handling of simple problems. The longer completion times for specific tasks in C and Erlang may indicate the need for optimization. However, the quick resolution of these tasks highlights the straightforward nature of the problems and the AI’s ability to navigate simple algorithmic challenges easily. The following tables illustrate ChatGPT’s runtime performance and memory efficiency, relative to other developer’s submissions:

Task	Python	C	Java	Golang	Erlang
Two Sums	97.39%	81.16%	92.24%	93.89%	76.92%
Add Binary	99.62%	37.39%	61.62%	100.0%	0.0%
Remove Linked List Elements	46.70%	36.88%	91.36%	92.09%	100.0%
Implement Stack with Queue	48.39%	53.08%	100.0%	78.15%	0.0%
Concatenation of Arrays	96.19%	40.65%	96.48%	89.43%	100.0%

Table 6.4: Runtime Performance (Easy)

Table 6.4 compares ChatGPT’s runtime performance for “Easy” level tasks across various programming languages. Each task’s percentage score indicates the proportion of LeetCode users that ChatGPT outperformed in the respective category. [28]

In Python, ChatGPT demonstrated high performance in tasks such as “Two Sums” and “Concatenation of Arrays”, achieving scores of 97.39% and 96.19% respectively. This

indicates that ChatGPT’s solutions outpaced the majority of submissions. However, in tasks like “Remove Linked List Elements” and “Implement Stack with Queue”, ChatGPT’s performance was relatively lower, although it still outperformed over 46.70% and 48.39% of users, respectively. This data highlights areas where further runtime optimization may be pursued to enhance ChatGPT’s efficiency in generating solutions.

Task	Python	C	Java	Golang	Erlang
Two Sums	99.39%	95.75%	5.0%	18.68%	53.85%
Add Binary	94.65%	91.12%	49.65%	21.58%	0.0%
Remove Linked List Elements	92.16%	90.01%	54.71%	31.03%	100.0%
Implement Stack with Queue	91.21%	94.52%	8.48%	26.05%	0.0%
Concatenation of Arrays	88.11%	88.35%	25.12%	49.08%	100.0%

Table 6.5: Memory Efficiency (Easy)

Table 6.5 illustrates the memory usage analysis for ChatGPT’s performance of the same tasks outlined in Table 6.4. Memory efficiency is critical as it indicates the optimization of the solution concerning space complexity. This optimization is important in resource-constrained environments and large-scale applications.

In various tasks such as “Two Sums” and “Concatenation of Arrays” in Python, ChatGPT has demonstrated notably high memory efficiency, scoring 98.76% and 99.63%. However, “Add Binary” task in Java posed a challenge, with a lower score of 32.45%, indicating an area where memory optimization could be beneficial. The varying memory efficiency across different tasks and languages provides valuable insights into the trade-offs between runtime performance and memory usage.

Although our results show a 100% rating in both performance and memory efficiency for Erlang, it is important to note that this percentage is based on only two submissions. Consequently, the high efficiency and performance metrics should be interpreted cautiously due to the limited sample size, which may not provide a representative comparison.

In contrast, Python, with 1,102 submissions, demonstrates the impact of a larger dataset, where achieving a high percentage is more challenging due to the larger comparison base. This disparity in sample sizes highlights the role of submission volume in assessing performance and memory efficiency metrics across programming languages. Therefore, while Erlang’s results are notable, the broader range of submissions in languages like Python provides a more robust basis for evaluation.

The detailed descriptions provide a thorough comprehension of the data presented in the tables, allowing the reader to comprehend ChatGPT’s performance in terms of coding efficiency. By assessing runtime and memory usage, we illustrate the AI’s problem-solving capabilities compared to the developer community.

The findings outlined in this subsection are important for understanding ChatGPT’s capabilities and limitations in tackling “Easy” tasks. The comprehensive analysis of the solved tasks, test cases passed, time tracking, and performance metrics offers valuable insights that could potentially guide future research and development.

6.2 Results (Medium)

This subsection provides a comprehensive overview of the research findings of “Medium” difficulty tasks and their implications. Through the examination of these tasks, we aim to gain valuable insights into the effectiveness and adaptability of the methods utilized. The number of test cases passed serves as a benchmark for the reliability and correctness of the provided solutions, offering a comprehensive assessment of ChatGPT’s performance.

The following table provides an overview of the solved tasks:

Task	Python	C	Java	Golang
Array with Stack Operations	✓	✓	✓	✓
Validate IP Addresses	✓	✓	✓	✓
Convert Array into 2D Array	~	×	~	~
Total Distance Traveled	✓	✓	✓	✓

Table 6.6: Solved Tasks (Medium)

Table 6.6 shows that ChatGPT successfully solved most tasks, except “Convert Array into 2D Array”. For this task, two different entries are provided, “×” and “~”. The “×”, recorded for “C”, signifies that ChatGPT failed to generate a solution that passed the predefined test cases, declaring the task unsuccessful. In contrast, the “~” indicates that ChatGPT successfully produced a solution that passed all test cases. However, this outcome was achieved only with substantial external assistance, such as detailed or advanced prompts, highlighting its reliance on additional guidance to produce the correct solution. The following table provides an overview of the number of tests passed for each task:

Task	Python	C	Java	Golang
	Test	Tests	Tests	Tests
Array with Stack Operations	49/49	49/49	49/49	49/49
Validate IP Addresses	74/74	74/74	74/74	74/74
Convert Array into 2D Array	1035/1035	0/1035	1035/1035	1035/1035
Total Distance Traveled	10002/10002	10002/10002	10002/10002	10002/10002

Table 6.7: Tests Passed (Medium)

Table 6.7 shows the number of tests each task passed. Notably, every task achieved a perfect score across all languages, with 49/49 tests passed for the “Array with Stack Operations”, 74/74 for “Validate IP Addresses”, and an impressive 1035/1035 for “Convert Array into 2D Array”. “Total Distance Traveled” also shows a perfect score with 10002/10002 tests passed. This consistency indicates a high level of code accuracy and suggests that the implemented solutions are robust across various test cases and scenarios.

The following table provides an overview of the time, in minutes, ChatGPT needed to solve each task:

Task	Python	C	Java	Golang
	Time	Time	Time	Time
Array with Stack Operations	14 min	35 min	20 min	5 min
Validate IP Addresses	15 min	35 min	24 min	2 min
Convert Array into 2D Array	30 min	30 min	30 min	30 min
Total Distance Traveled	8 min	30 min	20 min	15 min

Table 6.8: Time Tracking (Medium)

Table 6.8 documents the time to solve each task. Notably, ‘Array with Stack Operations’ took the longest in C at 35 minutes, while Golang completed the task in just 5 minutes, indicating significant differences in efficiency among the languages. Similarly, ‘Validate IP Addresses’ was completed most quickly in Golang at 2 minutes, compared to 35 minutes in C. The duration of solving ‘Convert Array into 2D Array’ was consistently about 30 minutes. ‘Total Distance Traveled’s’ time varied, with Golang only requiring 15 minutes compared to C at 30 minutes. These timings offer valuable insights into the performance and speed of different programming languages when handling similar tasks.

The following tables illustrate ChatGPT’s runtime performance and memory efficiency, relative to other developer’s.

Task	Python	C	Java	Golang
Array with Stack Operations	34.62%	100.0%	100.0%	100.0%
Validate IP Addresses	50.39%	100.0%	27.60%	56.67%
Convert Array into 2D Array	58.56%	0.0%	86.63%	82.44%
Total Distance Traveled	61.29%	41.51%	100.0%	7.69%

Table 6.9: Runtime Performance (Medium)

The 0.0% score for ‘Convert Array into 2D Array’ in Table 6.9 is due to ChatGPT generating a solution that turned out to be a plagiarized version of one of the correct options we provided. The other solutions provided were either completely incorrect or repeats of the initial incorrect solutions provided by ChatGPT.

For the task ‘Array with Stack Operations’, ChatGPT’s Python solution outperformed 34.62% of other developers, while its C and Java solutions excelled, surpassing 100%. This performance gap suggests that certain programming languages may offer a suitable environment for optimizing runtime in specific tasks. ChatGPT’s Python solution for ‘Convert Array into 2D Array’ showed moderate efficiency, outperforming 58.56% of developers, while the Java solution beat 86.63%. It is important to note that these performance metrics are highly dependent on the solutions provided by other users. Specifically, a lack of solutions in a certain programming language can result in a higher probability of ChatGPT outperforming the user and achieving great performance metrics.

Task	Python	C	Java	Golang
Array with Stack Operations	96.15%	95.0%	7.30%	16.0%
Validate IP Addresses	94.49%	84.62%	29.69%	40.0%
Convert Array into 2D Array	95.99%	0.0%	26.84%	53.77%
Total Distance Traveled	95.16%	100.0%	59.38%	98.08%

Table 6.10: Memory Efficiency (Medium)

Table 6.10 illustrates ChatGPT’s memory efficiency in “Medium” difficulty tasks. For the task “Validate IP Addresses”, ChatGPT’s Python implementation was more memory-efficient than 50.39% of other developers, showing a balanced use of memory resources. In contrast, the Java solution for the same task demonstrated a lower memory efficiency, outperforming only 27.60%, suggesting potential areas for memory optimization. The Python solution of “Total Distance Traveled” outperformed 61.29% of submissions, whereas the Golang solution showed room for improvement, with a lower efficiency that outperformed 7.69%.

When comparing Python and C for the “Validate IP Address” task, clear differences in performance and memory efficiency are observed. The number of submissions for Python, with 177, and C, with 61, are not identical but are sufficiently close, allowing for a more balanced comparison than in other tasks. This makes the results more meaningful when evaluating how ChatGPT’s solution in each language performs in processing-intensive tasks.

In terms of performance, C achieves a perfect performance score of 100, demonstrating its superior speed for processing this task. In contrast, Python has a performance rating of 50.39%, indicating lower efficiency in execution. However, Python outperforms C regarding memory efficiency, with a rating of 94.49%, while C achieves 84.62%. This shows that Python is more efficient in memory usage, which could make it preferable in scenarios with limited resources, despite its lower performance.

Similarly, when comparing Java and Python for the “Convert Array into 2D Array” task, Java shows superior performance, with a rating of 86.63% compared to Python’s 58.56%. However, Python excels in memory efficiency, scoring 95.99%, whereas Java has a much lower memory efficiency of 26.84%. Both languages have substantial submission counts, with Java having 744 submissions and Python having 621, making them statistically significant for comparison. This contrast highlights the trade-off between processing speed and memory usage, with Java being more performance-oriented and Python better suited for memory-constrained environments.

These comparisons illustrate how the programming language can impact the execution speed and resource consumption of ChatGPT’s solutions.

In summary, the results presented in this section contribute to our understanding of ChatGPT’s performance and efficiency and can guide future research. Additionally, the detailed analysis of the tasks completed, the number of test cases passed, time tracking and performance metrics provide valuable insights for decision-making and further advancements in the field.

6.3 Results (Hard)

This section offers a comprehensive overview of the research findings and their implications for “Hard” tasks. It presents a thorough analysis, focusing on various aspects such as task completion, test case success rates, time tracking, and performance metrics. By evaluating the tasks successfully solved by ChatGPT, we can assess the effectiveness and adaptability of the methods employed, with the number of test cases passed serving as an indicator of solution reliability and accuracy.

Furthermore, this section includes a detailed time-tracking analysis, similar to the other subsections. This analysis allows for an assessment of the efficiency of the problem-solving strategies used by ChatGPT. Additionally, the runtime and memory performance metrics offer valuable insights into the overall efficiency of ChatGPT compared to other Leetcode users. [28]

The following table provides an overview of the solved tasks:

Task	Python	Golang	Java
Encrypt/Decrypt Strings	~	~	~
Regex Matching	✓	✓	✓
Sudoku Solver	✓	✓	✓

Table 6.11: Solved Tasks (Hard)

Table 6.11 presents data on complex tasks categorized as “Hard”. The results demonstrate that the majority of tasks were successfully solved across the three programming languages, highlighting the effectiveness of the algorithms and problem-solving approaches in addressing advanced challenges. The entries for “Encrypt/Decrypt Strings”, marked as “~”, indicate that ChatGPT solved the task but required substantial assistance. Specifically, we provided functional solutions in each programming language, enabling ChatGPT to analyze and refine the required functionality in greater detail.

The following table provides an overview of the number of tests passed for each task:

Task	Python	Golang	Java
	Tests	Tests	Tests
Encrypt/Decrypt Strings	211/211	211/211	211/211
Regex Matching	355/355	355/355	355/355
Sudoku Solver	6/6	6/6	6/6

Table 6.12: Tests Passed (Hard)

Table 6.12 illustrates the test pass rates for the challenging tasks. “Encrypt/Decrypt Strings” and “Regex Matching” achieved perfect scores by passing 211/211 and 355/355 tests, respectively, demonstrating the exceptional performance of the solutions. Additionally, “Sudoku Solver”, despite having a smaller number of tests, achieved a flawless pass rate of 6/6. This outstanding success across all tasks and languages highlights the technical expertise in handling “Hard” challenges.

The following table provides an overview of the time tracking for each task:

Task	Python	Golang	Java
	Time	Time	Time
Encrypt/Decrypt Strings	100 min	120 min	120 min
Regex Matching	2 min	2 min	2 min
Sudoku Solver	2 min	2 min	2 min

Table 6.13: Time Tracking (Hard)

The times to solve the tasks, documented in Table 6.13, show that Python consistently demonstrates the quickest completion times, finishing each task within 2 minutes, except for “Encrypt/Decrypt Strings”, with 100 minutes. Notably, this task took the longest in every programming language, with Python being the programming language with the fastest time to attain a satisfactory solution from ChatGPT. Based on the observed variation of time across tasks in the results, there is a need for optimization for specific computational tasks especially regarding security-related topics.

The following tables illustrate ChatGPT’s performance metrics compared to other developers.

Task	Python	Java	Golang
Encrypt/Decrypt Strings	25.0%	96.40%	100.0%
Regex Matching	84.08%	100.0%	65.54%
Sudoku Solver	31.61%	81.28%	62.21%

Table 6.14: Runtime Performance (Hard)

ChatGPT’s Python runtime performance was moderate for “Encrypt/Decrypt Strings”, achieving a runtime surpassing 25.0% of other users. On the other hand, the Java and Golang solutions exhibited exceptional efficiency, outperforming 96.40% and 100.0%, respectively. For “Regex Matching”, the Python implementation showed high efficiency, surpassing 84.08% of submissions. Java performed exceptionally as well with a success rate of 100.0%, while Golang exhibited relatively lower efficiency at 65.54%, indicating potential areas for runtime performance enhancement for ChatGPT’s solutions in these languages.

“Sudoku Solver” demonstrated different performances across programming languages, with Python solving faster than 31.61% of developers, Java outperforming 81.28%, and Golang surpassing 62.21%. These statistics highlight ChatGPT’s strengths and weaknesses in each language in handling complex algorithmic problems efficiently and underscore the significance of choosing the appropriate programming language based on the specific computational demands of the task.

Task	Python	Java	Golang
Encrypt/Decrypt Strings	100.0%	83.88%	70.0%
Regex Matching	90.46%	53.19%	99.74%
Sudoku Solver	94.43%	59.87%	50.0%

Table 6.15: Memory Efficiency (Hard)

ChatGPT’s Python solution for “Encrypt/Decrypt Strings” demonstrated optimal memory usage, achieving a perfect score with 100.0% efficiency. The Java and Golang implementations also performed well, with 83.88% and 70.0% efficiency, respectively. In the case of “Regex Matching”, Python and Golang exhibited impressive memory efficiency, outperforming 90.46% and 99.74% of submissions, respectively. In contrast, the Java solution showed lower efficiency at 53.19%, indicating the potential for memory usage refinement. Regarding “Sudoku Solver”, Python showcased high memory efficiency at 94.43%, outperforming most other developers. On the other hand, Java and Golang had lower scores, with 59.87% and 50.0% efficiency, respectively.

When comparing ChatGPT’s solution for Python and Golang for “Encrypt/Decrypt Strings”, the differences in performance and memory efficiency reveal distinct strengths for each language. Golang achieves a perfect score of 100% in performance, suggesting that it excels in handling encryption and decryption operations efficiently. This makes Golang an excellent choice for scenarios where speed is crucial. In contrast, Python shows a much lower performance rating of 25.0%, indicating that it may not be as optimized for the fast execution of this task.

On the other hand, the Python solution outperforms Golang in terms of memory efficiency, surpassing 100% of user submissions. This indicates that ChatGPT utilizes memory more effectively in Python, making it a better option where minimizing resource consumption is a priority. Golang, while impressive in performance, shows a memory efficiency of 70.0%, suggesting that it consumes more memory to achieve its faster execution.

These performance metrics provide valuable insights into how effectively ChatGPT’s solutions utilize system resources. This is crucial for the performance of software applications, especially in the context of complex tasks. It underscores the significance of optimizing memory usage to ensure accurate, quick, and resource-efficient solutions.

The detailed analysis of runtime and memory efficiency highlights areas of excellence and identifies opportunities for further optimization, providing a comprehensive view of ChatGPT’s capabilities in solving “Hard” difficulty programming challenges.

6.4 Results (Very Hard)

This section focuses on our research findings on the most challenging tasks. In contrast to the other subsections, the evaluation of these tasks did not involve test cases due to their complexity. Instead, we used a manual testing approach to ensure the solutions were theoretically sound and practically applicable. This manual testing approach validates the functionality and effectiveness of the solutions in real-world scenarios, which automated tests might not fully capture.

We begin by examining the successful resolution of these highly challenging tasks. This analysis provides valuable insights into the effectiveness and adaptability of ChatGPT’s

methodologies when confronted with tough challenges. Additionally, our findings include a detailed breakdown of the time taken to address each task, offering a comprehensive overview of the duration of the problem-solving process.

The following table provides an overview of the solved tasks:

Task	Python
AI creates Assistance AI	~
Arcade Games App	~
EvilTwin Wi-Fi	~

Table 6.16: Solved Tasks (Very Hard)

In Table 6.16, the “~” symbol indicates that the most challenging tasks in the experiment were completed using a collaborative, pair programming approach. This method accomplished all tasks, showcasing the effective application of advanced techniques and problem-solving strategies to address complex issues. This success highlights the potential for ChatGPT to contribute to tasks traditionally seen as too difficult for automated systems when working in collaboration with human users.

The following table provides an overview of the testing method applied:

Task	Python
	Tests
AI creates Assistance AI	Manually Tested
Arcade Games App	Manually Tested
EvilTwin Wi-Fi	Manually Tested

Table 6.17: Tests Passed (Very Hard)

Table 6.17 illustrates that we performed manual testing, as no automated test suite was available for this set of challenges. Instead, we used predefined acceptance criteria to evaluate the solutions. These criteria were carefully defined to cover key functional and performance aspects of each task. ChatGPT met all the criteria, demonstrating its ability to solve these tasks according to the specified requirements. This manual evaluation highlights the reliability and accuracy of ChatGPT’s solutions in meeting the expected outcomes.

The following table provides an overview of the time, in minutes, ChatGPT needed to solve each task:

Task	Python
	Time
AI creates Assistance AI	5 h
Arcade Games App	12 h
EvilTwin Wi-Fi	64 h

Table 6.18: Time Tracking (Very Hard)

Table 6.18 illustrates the time investment required to solve “Very Hard” tasks in Python.

The development of the “Assistance AI” took 5 hours, signifying a considerable effort in creating an AI capable of assisting with complex tasks. Creating the “Arcade Games App” required 12 hours, reflecting the detailed process of developing an application with multiple interactive elements. The most time-consuming task was “EvilTwin Wi-Fi”, taking 64 hours to complete, indicating the possible involvement of sophisticated algorithms and security measures to address or simulate network vulnerabilities. These time frames demonstrate these tasks’ complexity and labor-intensive nature and underscore the varying time requirements that different complex tasks may demand. This insight could be pivotal for future research project planning and resource allocation.

The research outcomes presented in this subsection provide a comprehensive insight into ChatGPT’s capabilities in solving complex programming challenges. The successful completion of these tasks showcases the potential for AI to address intricate problems that are typically considered beyond the reach of conventional automated systems. Furthermore, the assessments from our manual testing and time-tracking analysis provide valuable perspectives on the challenges encountered and their effectiveness.

6.5 Distribution of Prompt Variations Across Tasks

Table 6.19 provides a detailed overview of the distribution of prompt variations across task difficulties, highlighting the strategies used to evaluate ChatGPT’s performance. These variations reflect the increasing complexity of tasks and the corresponding levels of assistance required by ChatGPT to achieve satisfactory results.

The “Standard Prompt” consisted of the basic prompt 5.2 followed by the task description. ChatGPT demonstrated a high degree of autonomy in solving tasks with this prompt, particularly for “Easy” tasks, where it successfully addressed challenges with minimal guidance. Notable exceptions included the “Two Sums” task in Erlang, which required additional input. Interestingly, the “Standard Prompt” sufficed for the “Medium” difficulty task “Validate IP Addresses” in Golang, and for the “Hard” difficulty tasks “Sudoku Solver” and “Regex Matching” across all programming languages.

The “Standard Prompt + Additional Information” expanded on the “Standard Prompt” by clarifying task requirements or constraints and providing examples. This variation was particularly effective for “Medium” difficulty tasks, where the added context often bridged the gap between task complexity and ChatGPT’s capabilities. For example, the “Two Sums” task in Erlang required this additional guidance despite being labeled as “Easy”.

“Advanced Prompts” were applied to two tasks: one categorized as “Hard” and one as “Medium”, across all programming languages. These tasks were characterized by their complexity and potential misalignment with the requirements of the LeetCode test suite. Detailed instructions, including step-by-step descriptions of expected code behavior, were provided to address this, as shown in Listing 7.1.4. This structured guidance underscored the importance of precision in solving highly complex programming challenges.

A collaborative approach was adopted for “Very Hard” tasks. This method simulated a pair programming environment, engaging ChatGPT in iterative exchanges to refine solutions progressively. The dialogic process facilitated adaptability and precision, proving critical for tackling the most complex problems.

Despite these tailored strategies, ChatGPT failed three tasks: “Add Binary” in Erlang,

“Implement Stack with Queue” in Erlang, and “Convert Array into 2D Array” in C. These failures highlight areas where even advanced prompts and collaboration were insufficient, offering valuable insights for refining prompt design and AI training.

Table 6.19 summarizes how these prompt variations were distributed across task difficulties. The observed progression underscores a direct correlation between task difficulty and the level of assistance required. While simpler tasks were addressed with minimal intervention, increasing complexity necessitated a more tailored and iterative approach. This pattern highlights ChatGPT’s adaptability and the critical role of prompt customization in optimizing performance.

Prompt Type	Easy	Medium	Hard	Very Hard
Standard Prompt (SP)	22	1	6	0
SP + Additional Information	1	11	0	0
Advanced Prompt	0	3	3	0
Collaboration	0	0	0	3

Table 6.19: Prompt Variations Used by Difficulty

6.6 Summary

This chapter presented an in-depth analysis of ChatGPT’s performance across a spectrum of programming tasks, ranging from “Easy” to “Very Hard”. By leveraging LeetCode’s benchmarking tools, the experiments evaluated solution accuracy, runtime efficiency, and memory usage across Python, C, Java, Golang, and Erlang. These assessments allowed for a detailed comparison of ChatGPT’s computational capabilities compared to human developers at varying levels of expertise, providing a comprehensive view of the model’s strengths and areas for improvement.

The findings reveal distinct performance characteristics at each difficulty level:

- **Easy Tasks:** ChatGPT consistently demonstrated high accuracy, achieving a perfect or near-perfect test pass rate across all languages tested. Python and C showed exceptional results, with task completion times averaging 1–2 minutes. The model’s memory efficiency in these languages was also excellent, often outperforming over 90% of other LeetCode submissions. However, specific tasks in Erlang displayed comparatively higher runtimes, with completion times sometimes extending to 20-25 minutes. This variability suggests that while ChatGPT is well-suited for handling foundational tasks, its performance is contingent upon the chosen language, with Erlang posing occasional challenges in runtime efficiency.
- **Medium Tasks:** The model’s performance on “Medium” difficulty tasks highlighted its adaptability and the impact of language-specific efficiencies. For instance, in tasks such as “Array with Stack Operations” and “Validate IP Addresses,” Golang frequently achieved faster solution times, as low as 5 minutes, while C often required up to 35 minutes. Memory usage efficiency varied notably, with Python and Java delivering optimized solutions that outperformed a substantial percentage of submissions. These results underscore ChatGPT’s flexibility in managing moderate

complexity, yet they also suggest that programming language choice significantly influences performance efficiency and completion speed for tasks involving more sophisticated algorithms and data handling.

- **Hard Tasks:** ChatGPT’s accuracy remained high for tasks at this level, achieving nearly perfect pass rates across tasks like “Regex Matching” and “Sudoku Solver.” However, runtime performance exhibited substantial variation, with Java and Golang often providing faster solutions compared to Python, which lagged in specific high-demand tasks. For instance, in “Regex Matching”, ChatGPT’s Java solution outperformed 100% of other submissions, while Python was only moderately efficient, highlighting the model’s dependence on language-specific optimization. Memory usage efficiency also varied, with Java showing lower optimization in some cases, suggesting the need for tailored improvements in memory handling for more complex algorithms. These findings demonstrate that, while ChatGPT can manage complex programming tasks, further refinement is needed to meet the efficiency standards expected in human-generated solutions consistently.
- **Very Hard Tasks:** For tasks classified as “Very Hard”, ChatGPT required extensive manual intervention and significant time investments. For example, the “EvilTwin Wi-Fi” task demanded 64 hours of processing time, reflecting the complexity of security-related algorithms and high computational demands. Tasks such as “AI creates Assistance AI” and “Arcade Games App” also required multiple hours to solve, with variable success depending on the language and nature of the problem. These results indicate that ChatGPT’s current capabilities are limited in handling advanced, resource-intensive tasks requiring high accuracy levels and advanced computational processes. The extended time needed for these tasks in combination with the pair programming approach we applied for these tasks highlights the need for improvements in efficiency and algorithmic processing for complex applications.

The findings from these experiments suggest that ChatGPT is a viable tool for software engineering tasks, particularly at the “Easy” and “Medium” levels, where it approaches or exceeds human-level performance in accuracy and efficiency. Its test pass rates and consistent memory management demonstrate the model’s adaptability, especially in Python, which was observed to provide the most efficient memory usage across tasks. However, as task complexity increases, several critical limitations emerge, particularly regarding solution accuracy, runtime optimization, and memory handling, which vary significantly across programming languages.

The experimental data reveals that ChatGPT’s efficiency in solving “Easy” and “Medium” tasks position it as a useful assistant for handling routine programming problems, where developers can benefit from real-time, AI-driven solutions and explanations. However, for complex tasks that demand higher computational accuracy and resource optimization, such as “Regex Matching” or security-centric applications like “EvilTwin Wi-Fi”, the model’s current capabilities fall short. These limitations underscore the importance of optimizing ChatGPT for specific languages and contexts to maximize its potential as a software engineering tool.

Given these findings, future research should focus on several key areas:

- **Enhanced Runtime and Memory Efficiency:** Fine-tuning the model's handling of memory-intensive and time-sensitive tasks across languages, particularly for C and Java, would address the inefficiencies observed in “Medium” to “Very Hard” tasks. This could involve integrating language-specific optimization techniques or adjusting model parameters based on the programming language.
- **Scalability and Adaptability for Complex Tasks:** For high-complexity tasks, implementing targeted training or reinforcement learning methods that expose ChatGPT to complex data structures, algorithms, and performance benchmarks could improve its robustness. ChatGPT could become more reliable for applications requiring precision, such as cryptographic functions or high-performance computation by training on more diverse datasets that include advanced algorithms and language-specific idioms.
- **Application in Security and Resource-Constrained Environments:** The limitations seen in security-related tasks suggest that further model refinement is necessary for deploying ChatGPT in resource-intensive fields, such as cyber security. Integrating the model with real-world simulation environments for security tasks could provide an enhanced training ground, helping the model develop more accurate, memory-efficient, and responsive solutions.
- **Benchmarking Against Human Expertise:** Incorporating comparative studies with human developers across different skill levels could yield valuable insights into where AI can improve with human expertise most effectively. This would also inform areas where additional model tuning is required, such as for tasks where developers typically rely on language-specific optimizations or customized algorithms.
- **Broadening of Potential Use Cases:** The insights gained from these experiments suggest several practical applications, particularly in educational tools, automated testing, and basic code generation for industry-standard tasks. By enhancing ChatGPT's adaptability across programming languages and task complexities, the model could be a valuable assistant for experienced programmers and beginners aiming to automate simpler development tasks in environments like DevOps or data science.

In summary, while ChatGPT exhibits promising capabilities in addressing programming challenges, particularly those of low to “Medium” complexity, this study highlights areas for ongoing improvement to utilize its potential in software engineering fully. Future advancements in runtime and memory efficiency, complex task handling, and security-specific adaptations will be critical to positioning ChatGPT as a reliable tool for a broader spectrum of programming tasks. Such improvements could significantly impact its usability in professional and academic settings, contributing to more efficient workflows and expanding applications in programming environments.

Chapter 7

Discussion

This chapter provides an interpretative analysis of ChatGPT's performance in addressing software engineering tasks of varying complexity. Building on the empirical results in the preceding chapter, this chapter focuses on deeper patterns in ChatGPT's code generation strategies, response adaptability, and handling context-sensitive challenges across multiple programming environments. By documenting these outcomes, we gain insight into ChatGPT's strengths, such as efficiency and adaptability to task complexity, and limitations, particularly when navigating more advanced programming scenarios. These metrics are contextualized to provide a balanced view of ChatGPT's computational abilities. Additionally, this chapter assesses the quality and contextual relevance of ChatGPT's generated documentation, including user manuals and in-code comments. This aspect of the analysis highlights the model's potential to streamline documentation practices in software engineering, offering explanations that align with user needs and industry standards.

By critically evaluating ChatGPT's abilities, we aim to identify immediate improvements for AI-assisted programming and broader applications within software development. By isolating areas where ChatGPT demonstrates strong performance and those where refinement is necessary, this chapter aims to assess the model's reliability and potential as an AI-driven coding assistant, positioning it within the evolving landscape of AI-enhanced programming tools.

This discussion will transition into specific topics, such as the challenges associated with task complexities and the impact of programming language selection on performance. By systematically addressing these elements, the chapter contributes to a comprehensive assessment of ChatGPT's role and limitations in software engineering, providing data-driven insights that can guide future advancements in AI-assisted development.

7.1 Potential Relevance of ChatGPT in Software Engineering

The integration of ChatGPT into software engineering has the potential to impact the dynamics and efficiency of development processes significantly. In this section, we explore the potential relevance of ChatGPT in software engineering. ChatGPT can keep up with the latest advancements in programming languages, frameworks, design patterns, and software engineering methodologies using its continuous learning and adaptation capabilities. This adaptability allows it to be used for a broad spectrum of software engineering tasks.

Continuous learning and adaptation are pivotal in addressing the ongoing challenge of improving efficiency in software engineering. Rapidly evolving technologies and methodologies require developers to stay updated with the latest advancements. ChatGPT can keep up with the latest advancements by continuously providing and integrating new information. Furthermore, its ability to generalize from diverse data sources allows it to handle complex software engineering tasks.

In addition, ChatGPT showcases the potential for innovation in software engineering projects. It can streamline the development process, promote creativity and experimentation, and accelerate the translation of ideas into actionable code. Developers can use ChatGPT to prototype new features, explore alternative implementation strategies, and iterate on existing code with greater agility. This iterative approach to innovation allows developers to test theories and refine solutions rapidly.

In the following subsections, we go into more detail about ChatGPT's potential relevance in software engineering. Each subsection provides in-depth insights, examples, and recommendations, to assist in understanding the potential of ChatGPT in software engineering.

7.1.1 Continuous Learning and Adaptation

Improving efficiency is an ongoing challenge, particularly in rapidly evolving software engineering technologies and methodologies. One pivotal advance in recent years that addresses this challenge is the emergence of artificial intelligence models. ChatGPT can engage in meaningful dialogue and adapt to new information continuously. This subsection covers the implications of ChatGPT's capabilities for continuous learning and adaptation.

ChatGPT, using its underlying transformer architecture and large-scale pre-training, has a dynamic learning mechanism that enables it to integrate new knowledge iteratively. This characteristic allows ChatGPT to keep up with the latest advancements in programming languages, frameworks, design patterns, and software engineering methodologies. By continuously providing and integrating new information, ChatGPT can offer up-to-date insights and recommendations that reflect the current state-of-the-art software development.

Furthermore, ChatGPT's adaptability extends beyond familiarity with existing tools and techniques. Its ability to generalize from diverse data sources enables it to tackle a broad spectrum of software engineering tasks. This versatility is particularly advantageous in scenarios where conventional approaches may fail due to the lack of domain-specific expertise or the absence of established solutions. ChatGPT can be a versatile knowledge repository that rapidly retrieves relevant information.

In addition, ChatGPT's continuous learning capabilities facilitate its potential application as a collaborative partner in software development efforts. ChatGPT can refine its

understanding of technical concepts, industry trends, and user preferences through interactions with human developers and exposure to real-world software projects. ChatGPT’s continuous learning capabilities and adaptability to challenges can be seen in the following shortened versions of two attempts at “Add Binary” in C.

Attempt 1:

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 char* addBinary(char* a, char* b) {
5     int lenA = strlen(a);
6     int lenB = strlen(b);
7     int maxSize = lenA > lenB ? lenA : lenB;
8     // Allocate memory for the result string and an extra character for
9     // the potential carry
10    char* result = (char*)malloc((maxSize + 2) * sizeof(char));
11    result[0] = '0'; // Set the first character to '0'
12    result[maxSize + 1] = '\0'; // Null-terminate the result string
13    int carry = 0;
14    int i = lenA - 1, j = lenB - 1, k = maxSize;
15    while (i >= 0 || j >= 0 || carry > 0) {
16        int bitA = (i >= 0) ? a[i] - '0' : 0;
17        int bitB = (j >= 0) ? b[j] - '0' : 0;
18        int sum = bitA + bitB + carry;
19        result[k] = (sum % 2) + '0';
20        carry = sum / 2;
21        i--;
22        j--;
23        k--;
24    }
25    // If there is a carry after processing all bits, prepend it
26    if (carry > 0) {
27        result[0] = carry + '0';
28        return result;
29    } else {
30        // Skip the leading zeros in the result (if any)
31        int leadZeros = 0;
32        while (result[leadZeros] == '0' && result[leadZeros + 1] != '\0') {
33            leadZeros++;
34        }
35        return result + leadZeros;
36    }
37 }
```

Listing 7.1: Code generated by ChatGPT in the first attempt

Despite considerable interaction, ChatGPT did not deliver a satisfactory solution for the given task.

Attempt 2:

```
1 #include <stdio.h>
2 #include <string.h>
3
4 // Function to add two binary strings
5 char* addBinary(char* a, char* b) {
6     int len_a = strlen(a);
7     int len_b = strlen(b);
8
9     // Find the maximum length of the two strings and add 1 for the
        possible carry
10    int max_len = (len_a > len_b) ? len_a : len_b;
11    char* result = (char*)malloc((max_len + 2) * sizeof(char)); // +2 for
        sum and possible carry
12    result[max_len + 1] = '\0'; // Null-terminate the result string
13
14    int carry = 0;
15    int i = len_a - 1, j = len_b - 1, k = max_len;
16
17    // Perform binary addition
18    while (i >= 0 || j >= 0 || carry > 0) {
19        int sum = carry;
20
21        if (i >= 0) {
22            sum += (a[i] - '0');
23            i--;
24        }
25        if (j >= 0) {
26            sum += (b[j] - '0');
27            j--;
28        }
29
30        // Calculate the current bit in the result
31        result[k] = (sum % 2) + '0';
32        k--;
33
34        // Update the carry for the next iteration
35        carry = sum / 2;
36    }
37
38    // If there is no carry in the leftmost bit, shift the result to the
        right
```

```
39     if (result[0] == '0') {  
40         char* shifted_result = result + 1;  
41         free(result);  
42         return shifted_result;  
43     }  
44  
45     return result;  
46 }
```

Listing 7.2: Code generated by ChatGPT in the second attempt

On the second attempt, ChatGPT promptly provided an applicable solution upon receiving the task description, without requiring any additional input or interaction from us.

Specifically, this instance illustrates ChatGPT's initial struggle to solve the task, requiring significant time, interaction, additional information, and assistance. On the second attempt, which we started after one week, ChatGPT solved the task immediately without needing further information or help from our side. Furthermore, ChatGPT provided more efficient code than in the first attempt, which shows the potential of ChatGPT to learn and improve by acquiring information from user interactions. ChatGPT can augment human expertise with its vast knowledge repository in a collaborative learning process. This promotes a symbiotic relationship, benefiting from human feedback to enhance its performance and relevance in software engineering contexts.

Notably, the adaptive nature of ChatGPT highlights its potential flexibility for dynamic challenges encountered during software development. As projects evolve, requirements shift, and technologies evolve, ChatGPT can quickly re-calibrate its recommendations and solutions to align with requirements and constraints. This agility mitigates the risk of obsolescence and ensures that ChatGPT remains a valuable asset throughout the software development life cycle.

In summary, the continuous learning and adaptation capabilities of ChatGPT represent a potential change in software engineering. By leveraging its ability to gain new knowledge iteratively, generalize across diverse domains, collaborate with human developers, and adapt to evolving circumstances, ChatGPT can enhance efficiency and innovation in software development endeavors.

7.1.2 Multi-Language Proficiency

The ability to assist in software engineering-related tasks in diverse programming languages is a valuable asset, facilitating collaboration, flexibility, and innovation. ChatGPT emerges as a helpful tool, characterized by its proficiency in multiple programming languages. This subsection illustrates the significance of ChatGPT's multi-language proficiency and its importance for software development.

The empirical findings of our research highlight ChatGPT's adaptability while solving coding challenges across various programming languages. ChatGPT efficiently solved challenges in programming languages such as Python, C, Java, Golang, and Erlang. This versatility allows developers to leverage ChatGPT's assistance across various coding tasks, regardless of the programming language used. ChatGPT adapts to developers' varied

needs and preferences, enhancing the accessibility of challenges associated with software engineering. Furthermore, the multi-language proficiency of ChatGPT extends beyond mere syntactic comprehension, including an understanding of language-specific libraries, frameworks, and idiomatic expressions. This enables ChatGPT to provide contextually relevant recommendations and solutions that align with each programming language’s unique characteristics and requirements.

A noticeable observation from this empirical research is ChatGPT’s exceptional proficiency in Python, a widely adopted programming language renowned for its simplicity, readability, and versatility. ChatGPT excels in addressing coding challenges posed in Python, showcasing an understanding of Python idioms, syntax, and best practices. Incorporating a Python interpreter enhances ChatGPT’s capabilities, enabling seamless execution and assessment of Python code snippets within the conversational interface. Consequently, developers working on Python projects can rely on ChatGPT as a suitable tool, offering guidance and solutions.

ChatGPT’s Python proficiency became evident in our research by solving challenges in Python that failed in alternative programming languages, such as the “Arcade Games App” in Flutter. This underscores its understanding of Python’s semantics and problem-solving paradigms. More precisely, in our first attempt, we tried to implement an arcade games application in Flutter with ChatGPT, which failed because ChatGPT can not provide a complete project structure in Flutter due to the rather extensive project structure, which can not be covered in ChatGPT’s response. This would result in the user having to handle all the other existing issues due to missing adaptations in the provided basic structure given by Android Studio. Furthermore, the adaptations in Flutter were incorrect in certain instances, showing a missing understanding of the relatively new programming language, compared to the more explored programming language used in our second try, Python. This research result highlights ChatGPT’s expertise in widely-used programming languages such as Python, as displayed by its success in solving the task using Python, unlike in Flutter. Despite Flutter being a more appropriate choice for developing an arcade game application, a fact acknowledged by ChatGPT itself, ChatGPT struggled with Flutter. This can be seen in the following example from our dialogue with ChatGPT.

User:

Which programming language would you recommend for creating a simple Arcade Game App from which the user can choose three games?

ChatGPT:

If you want to create an app that allows users to choose from three simple arcade games, you would typically need to use a mobile app development framework. Here are some popular options depending on your expertise and platform preferences:

1. **React Native:** React Native is a framework that allows you to build cross-platform mobile apps using JavaScript and React. It is a good choice if you want to target both iOS and Android with a single codebase.
2. **Flutter:** Flutter is an open-source UI software development kit created by Google. It uses the Dart programming language and allows you to build natively compiled applications for mobile, web, and desktop from a single codebase.
3. **Swift (iOS) and Kotlin (Android):** If you prefer native development, you can use Swift for iOS and Kotlin for Android. Both are the official programming languages for their respective platforms.

Once you have chosen the framework, you can proceed with the following steps:

1. **Design the User Interface (UI):** Create a user-friendly interface that allows users to choose from the three arcade games. This could be a simple menu or a carousel-style interface where users can swipe or tap to select a game.
2. **Develop the Arcade Games:** Depending on the chosen platform and language, implement the three simple arcade games. You can use the programming environments/languages mentioned earlier (e.g., Python with Pygame, JavaScript with Phaser) to create the games.
3. **Integrate the Games with the UI:** Connect the selected games to the UI elements so that when the user chooses a game, they are taken to the corresponding game screen.
4. **Test and Debug:** Thoroughly test the app on various devices and emulators to ensure it works as expected. Debug any issues that arise during testing.
5. **Publish the App:** Once you are satisfied with the app's functionality and it has been thoroughly tested, you can publish it to the respective app stores (Google Play Store for Android and the Apple App Store for iOS) or other distribution channels.

Remember to keep the games simple and engaging for a smooth user experience. Also, ensure that you comply with the app store guidelines and policies during the development and publishing process.

User:

What would you recommend?

ChatGPT:

As of my last update in September 2021, I would recommend using Flutter for creating an app that allows users to choose from three simple arcade games. Flutter is an excellent choice for several reasons:

1. Cross-Platform Development: Flutter allows you to write code once and deploy it on both iOS and Android platforms. This saves development time and effort compared to building separate native apps for each platform.
2. Fast Development and Hot Reload: Flutter's hot reload feature enables you to see changes instantly during development, making the iteration process faster and more efficient.
3. Rich Set of Widgets: Flutter offers a wide range of customizable widgets to build beautiful and responsive user interfaces, which is essential for creating engaging arcade games.
4. Performance: Flutter's code is compiled to native ARM code, resulting in high-performance apps that run smoothly on both iOS and Android devices.
5. Growing Community and Ecosystem: Flutter has a vibrant and growing community, which means you can find plenty of resources, plugins, and packages to enhance your app's functionality.
6. Dart Language: Although not as widely used as some other languages, Dart is easy to learn and use. Its modern features and clean syntax make it a suitable language for both beginners and experienced developers.

However, it's important to note that the tech landscape evolves rapidly, and new frameworks and tools may have emerged since my last update. Always consider the latest developments, community support, and your team's expertise before making a final decision. If possible, research the latest information about Flutter and other frameworks to ensure it still aligns with your project's requirements.

Furthermore, ChatGPT seems to encounter challenges when dealing with less popular programming languages such as Erlang, as indicated by the outcomes of the "Easy" tasks, particularly "Add Binary" and "Implement Stack with Queue", where its performance was lacking exclusively in the Erlang tasks. Despite receiving significant assistance for these tasks, ChatGPT proved ineffective in solving them. Additionally, ChatGPT generated libraries that resembled those used in different programming languages. However, these libraries were modified to suit Erlang, which initially appeared to offer a correct solution. Nonetheless, they ultimately failed to compile or function properly.

In conclusion, ChatGPT's multi-language proficiency indicates new possibilities in software engineering. Furthermore, ChatGPT emerges as a useful tool for developers seeking support in their coding challenges by demonstrating programming capabilities in various programming languages, with proficiency in Python, and showcasing adaptability.

7.1.3 Collaborative Coding

Effective collaboration is fundamental to driving innovation, enhancing productivity, and ensuring the delivery of high-quality software solutions. In this context, ChatGPT's capability to facilitate collaborative coding and streamline the development process can be beneficial. This subsection covers ChatGPT's role in promoting collaborative coding practices and potential contributions to software engineering projects.

One of the key advantages of ChatGPT in collaborative coding is its ability to accelerate prototyping and code generation. As highlighted by the findings in Section 6, ChatGPT excels in providing simple examples and code snippets. For instance, developers can provide a vague concept or a specific problem they are trying to solve, and ChatGPT can generate relevant code snippets or suggest approaches to solve the problem. This saves time and serves as a starting point for developers to build on. In more complex tasks, it can provide outlines or basic structures, address queries, and offer snippets that extend the project's functionality. For example, if a developer wants to implement a feature for user authentication, ChatGPT can provide the basic code structure and guide the developer on how to handle different scenarios such as user registration, login, and password reset. By eliminating the need to start from scratch, the development process is enhanced by providing developers with a basic framework. This can speed up the realization of ideas, facilitate fast prototyping, and simplify task implementations.

ChatGPT's role as a collaborative coding tool is particularly advantageous, by offering basic code structures and rapid prototyping. This capability enables developers to iterate on ideas and explore different implementation strategies. Consequently, the pace of development accelerates, enabling the potential for experimentation and innovation within software engineering projects. Notably, it serves as a valuable tool for rapid prototyping and the exchange of code snippets, especially in the early stages of a project.

It is worth noting that the quality of solutions provided by ChatGPT is highly dependent on the quality of user input, a topic we will examine further in the subsequent subsection. Clear and detailed requirements and descriptions are more likely to acquire satisfactory solutions. Thus, in a collaborative environment, ChatGPT is a useful assistant supporting the development process.

Beyond code generation, ChatGPT's collaborative capabilities extend to other aspects of software engineering, including the development of test cases, testing strategies, and quality assurance. Testing is a vital aspect of software development, and using its natural language processing capabilities, ChatGPT aids in formulating comprehensive test cases and devising effective testing approaches. For instance, developers can consult ChatGPT to generate test cases for specific modules or functionalities, ensuring comprehensive coverage and identifying potential edge cases. Additionally, it can help in developing testing strategies by providing insights on how to prioritize testing efforts, design test suites, and automate certain testing tasks. Consequently, it contributes to sustaining quality assurance processes, ultimately enhancing the overall software quality. By contributing to the qual-

ity assurance process, ChatGPT ensures that software solutions meet the requirements, leading to more robust and reliable software solutions.

ChatGPT's collaborative coding capabilities have the potential to significantly impact software development practices. By promoting consistency, facilitating information exchange, enabling rapid prototyping, and contributing to quality assurance efforts, ChatGPT emerges as a useful tool in the pursuit of innovation and improving efficiency in software engineering. Furthermore, it can serve as a valuable resource for learning and skill development. Developers can interact with ChatGPT to gain insights into best coding practices, explore new programming languages or frameworks, and seek guidance on complex coding concepts. This continuous learning and knowledge-sharing process ultimately strengthens the expertise of individual developers and the collective capabilities of the team.

Finally, its ability to generate foundational code components and assist in various aspects of the development process makes it a useful tool for software engineering teams seeking to enhance productivity, enable collaboration, and deliver high-quality software solutions.

7.1.4 Natural Language Processing

The intersection of natural language processing (NLP) and artificial intelligence has led to improvements regarding innovation and efficiency. ChatGPT's capabilities in NLP facilitate the seamless translation of software requirements and specifications into functional code. This subsection describes ChatGPT's NLP capabilities and contributions to the software development life cycle.

ChatGPT's proficiency in natural language processing allows it to comprehend and articulate software requirements. ChatGPT can identify key information and gain valuable insight by analyzing and interpreting text descriptions, effectively bridging the gap between conceptualization and practical execution. This proves especially beneficial throughout different phases of development, as ChatGPT enables the transformation of theoretical concepts into coding solutions.

The ability of ChatGPT to precisely translate requirements into logic and code could be valuable in software development. In projects where requirements are clearly and comprehensively articulated, ChatGPT can significantly accelerate code generation, allowing developers to focus on more complex and challenging tasks. Moreover, the ability of ChatGPT to accommodate adjustments and modifications to existing code based on user requests allows collaboration, enabling developers to communicate specific requirements efficiently and iteratively refine their implementations.

Our empirical findings underscore the role of clear and detailed descriptions in obtaining more accurate solutions from ChatGPT. In our research we focused on the key aspects when building a prompt, covered in Section 5.2. Providing precise and comprehensive requirements and descriptions increases the likelihood of achieving a satisfactory solution with the desired functionality, clear structure, and good readability, maintainability, and scalability. This can be seen in the following comparison of the two approaches of the task "Encrypt and Decrypt String". This comparison shows ChatGPT's incapability to solve this task until we provide a prompt with a detailed description and requirements for the implementation.

Case 1: Without step-by-step description:

You are given a character array `keys` containing unique characters and a string array `values` containing strings of length 2. You are also given another string array `dictionary` that contains all permitted original strings after decryption. You should implement a data structure that can encrypt or decrypt a 0-indexed string.

A string is encrypted with the following process:

- a) For each character `c` in the string, we find the index `i` satisfying `keys[i] == c` in `keys`.
- b) Replace `c` with `values[i]` in the string.

Note that in case a character of the string is not present in `keys`, the encryption process cannot be carried out, and an empty string `""` is returned.

A string is decrypted with the following process:

- a) For each substring `s` of length 2 occurring at an even index in the string, we find an `i` such that `values[i] == s`. If there are multiple valid `i`, we choose any one of them. This means a string could have multiple possible strings it can decrypt to.
- b) Replace `s` with `keys[i]` in the string.

Implement the `Encrypter` class:

- > `Encrypter(char[] keys, String[] values, String[] dictionary)` initializes the `Encrypter`
- > `String encrypt(String word1)` encrypts `word1` with the encryption process described above and returns the encrypted string.
- > `int decrypt(String word2)` returns the number of possible strings `word2` could decrypt to that also appear in a dictionary.

Listing 7.3: Description provided to ChatGPT

```

1 import java.util.HashMap;
2 import java.util.HashSet;
3 import java.util.Map;
4 import java.util.Set;
5 public class Encrypter {
6     private final Map<Character, String> encryptionMap;
7     private final Map<String, Set<String>> decryptionMap;
8
9     public Encrypter(char[] keys, String[] values, String[] dictionary) {
10         encryptionMap = new HashMap<>();
11         decryptionMap = new HashMap<>();
12         for (int i = 0; i < keys.length; i++) {

```

```

13         encryptionMap.put(keys[i], values[i]);
14     }
15     for (String value : values) {
16         String key = getKeyByValue(values, value);
17         decryptionMap.computeIfAbsent(value, k -> new
18             HashSet<>()).add(key);
19     }
20     // Populate decryptionMap with dictionary values
21     for (String word : dictionary) {
22         for (int i = 0; i < word.length() - 1; i += 2) {
23             String substr = word.substring(i, i + 2);
24             decryptionMap.computeIfAbsent(substr, k -> new
25                 HashSet<>()).add(word);
26         }
27     }
28
29     public String encrypt(String word) {
30         StringBuilder encrypted = new StringBuilder();
31
32         for (char c : word.toCharArray()) {
33             if (encryptionMap.containsKey(c)) {
34                 encrypted.append(encryptionMap.get(c));
35             } else {
36                 return ""; // Unable to encrypt, return empty string
37             }
38         }
39         return encrypted.toString();
40     }
41
42     public int decrypt(String word) {
43         int count = 1; // Default to 1, as every word can decrypt to itself
44
45         for (int i = 0; i < word.length() - 1; i += 2) {
46             String substr = word.substring(i, i + 2);
47             Set<String> possibleDecryptions =
48                 decryptionMap.getOrDefault(substr, new HashSet<>());
49             count *= possibleDecryptions.size();
50         }
51         return count;
52     }
53
54     private String getKeyByValue(String[] values, String value) {
55         for (int i = 0; i < values.length; i++) {
56             if (values[i].equals(value)) {

```

```
56         return String.valueOf((char) ('a' + i));
57     }
58 }
59 return "";
60 }
61 }
```

Listing 7.4: Code generated by ChatGPT

The generated code reveals certain errors that need further analysis. The code is problematic as it is not functioning correctly, thus being ineffective for its intended purpose. Furthermore, the code is lengthy, which potentially impedes its comprehensibility. An additional concern is the missing explanatory comments within the code, making it difficult to understand the underlying logic or design choices. Considering these issues, it is recommended to thoroughly examine and debug the code, accompanied by a comprehensive refactoring effort to fix its flaws. Subsequently, adequate documentation should be provided, including clear and concise comments to explain the code's structure and purpose, thereby assisting in understanding and future maintenance of the code.

Case 2: With step-by-step description:

Implement the following steps based on the given structure and provide comments.

Class Definition and Constructor:

- > The Encrypter class has a member variable `key_value`, which is an array of strings to store the values corresponding to each character key.
- > It also has a `HashMap` named `encrypted_dictionary` to store the encrypted words and their frequencies.
- > The constructor `public Encrypter(char[] keys, String[] values, String[] dictionary)` takes three arrays as parameters:
 - * `keys`: An array of characters representing the keys.
 - * `values`: An array of strings representing the values corresponding to each key.
 - * `dictionary`: An array of strings representing words to be encrypted and added to the encrypted dictionary.
- > It initializes the `key_value` array by mapping each character key to its corresponding value in the `values` array.
- > It then iterates over the `dictionary` array, encrypts each word using the `encrypt` method, and adds the encrypted word to the `encrypted_dictionary` with its frequency.

Encryption Method:

- > The `encrypt` method takes a word (`word1`) as input and returns its encrypted form.

- > It uses the `key_value` array to map each character of the input word to a two-character encrypted representation.
- > If the mapping fails (e.g., if the input contains a character that doesn't have a corresponding value), it returns null.
- > Otherwise, it constructs the encrypted word and returns it as a string.

Decryption Method:

- > The `decrypt` method takes an encrypted word (`word2`) as input and returns its frequency in the `encrypted_dictionary`.
- > If the encrypted word is not present in the dictionary, it returns 0.

To implement and use this class, you can create an instance of `Encrypter` by providing arrays of keys, values, and a dictionary of words to be encrypted. After that, you can use the `encrypt` and `decrypt` methods as needed.

Structure:

```
class Encrypter {  
    public Encrypter(char[] keys, String[] values, String[] dictionary) {  
    }  
    public String encrypt(String word1) {  
    }  
    public int decrypt(String word2) {  
    }  
}
```

Listing 7.5: Step-by-step description provided to ChatGPT

```
1 import java.util.HashMap;
2 import java.util.Map;
3 class Encrypter {
4     // Member variables
5     private final String[] key_value; // Mapping of characters to values
6     private final Map<String, Integer> encrypted_dictionary; // Dictionary
7     // Constructor
8     public Encrypter(char[] keys, String[] values, String[] dictionary) {
9         key_value = new String[256]; // Assuming ASCII characters, array to
10        store mappings
11        encrypted_dictionary = new HashMap<>(); // Initialize HashMap to
12        store encrypted words and frequencies
13        // Initialize key_value array with key-value pairs
14        for (int i = 0; i < keys.length; i++) {
15            key_value[keys[i]] = values[i];
16        }
17        // Encrypt and populate encrypted_dictionary with frequencies
18        for (String word : dictionary) {
19            String encryptedWord = encrypt(word);
20            encrypted_dictionary.put(encryptedWord,
21            encrypted_dictionary.getOrDefault(encryptedWord, 0) + 1);
22        }
23    }
24    // Encryption method
25    public String encrypt(String word1) {
26        StringBuilder encrypted = new StringBuilder(); // StringBuilder to
27        efficiently build the encrypted word
28        // Map each character of the input word to its encrypted
29        representation
30        for (char c : word1.toCharArray()) {
31            if (key_value[c] == null) {
32                return null; // Unable to encrypt, return null
33            }
34            encrypted.append(key_value[c]);
35        }
36        return encrypted.toString();
37    }
38    // Decryption method
39    public int decrypt(String word2) {
40        return encrypted_dictionary.getOrDefault(word2, 0); // Return the
41        frequency of the encrypted word in the dictionary
42    }
43 }
```

Listing 7.6: Code generated by ChatGPT

By providing a detailed, step-by-step description of how we wanted the code to be implemented, we were able to achieve a significant improvement in its functionality. ChatGPT now returns a working solution immediately, without any errors or bugs. The new implementation is more efficient, resulting in faster execution and better performance. The code itself has been optimized and refactored to be shorter and clearer, making it easier to understand and maintain. Additionally, ChatGPT included comments throughout the code, as requested, to provide clear explanations of its structure and purpose. Overall, these changes have greatly enhanced the overall quality and usability of the code, compared to the previous version.

This observation underscores the importance of effective communication in software development, in which unambiguous requirements result in precise and robust code. By enhancing the clarity and comprehensibility of software requirements, ChatGPT facilitates a more streamlined and efficient development process, in which developers can communicate complex requirements concisely and elevate code quality from the beginning.

Thus, integrating ChatGPT's NLP capabilities into the software development life cycle holds immense potential for enhancing productivity, prompting collaboration, and elevating code quality. By allowing the translation of software requirements into executable code, promoting collaboration, and emphasizing the importance of clear and detailed communication, ChatGPT can be a helpful assistant in software engineering and the pursuit of innovation.

7.1.5 Code Analysis

In software engineering, understanding code written by others is a significant challenge for developers. With the rising complexity of code bases, developers often face difficulties understanding and interpreting algorithms and structures. Innovative solutions and integrating Artificial Intelligence (AI) tools, such as ChatGPT, can address this challenge by providing advanced analysis and comprehension capabilities. This subsection investigates the capabilities of ChatGPT in code analysis and explanation.

The advanced Natural Language Processing (NLP) architecture of ChatGPT offers a promising opportunity for code analysis and interpretation. Trained in a diverse set of programming languages, ChatGPT exhibits a versatile understanding of code structures and patterns, making it a valuable asset for developers.

ChatGPT's proficiency in code analysis is highlighted by its ability to dissect code snippets, detect underlying patterns, and comprehend the logic inherent. A distinctive feature of ChatGPT lies in its capability to provide detailed explanations for code segments. This extends beyond simply identifying variables and functions, covering a profound breakdown of the provided code. More precisely, it can break down the code into manageable components, giving insight into the underlying processes and functionalities.

This can be seen in the following example, where ChatGPT analyzed the given code and provided a step-by-step explanation of how the code is structured and can be implemented following the provided steps.

```
1 Explain the following Golang code to me and provide a step-by-step
   explanation of how to implement it.
2
3 type Encrypter struct {
4     pairs map[byte]string
5     count map[string]int
6 }
7
8 func Constructor(keys []byte, values []string, dictionary []string)
   Encrypter {
9     pairs := make(map[byte]string)
10    for i, key := range keys {
11        pairs[key] = values[i]
12    }
13
14    result := Encrypter{
15        pairs: pairs,
16    }
17
18    result.count = make(map[string]int)
19    for _, word := range dictionary {
20        result.count[result.Encrypt(word)]++
21    }
22
23    return result
24 }
25
26 func (this *Encrypter) Encrypt(word1 string) string {
27     result := ""
28     for _, l := range word1 {
29         result += this.pairs[byte(l)]
30     }
31     return result
32 }
33
34 func (this *Encrypter) Decrypt(word2 string) int {
35     return this.count[word2]
36 }
```

Listing 7.7: Code of the task Encrypt/Decrypt Strings provided to ChatGPT

1 This Go code defines an ‘Encrypter’ struct and associated methods to
 perform encryption and decryption based on a provided set of keys and
 values. Let us **break** down the code and understand it step by step:

2
 3 1. Encrypter Struct:

```
4 type Encrypter struct {
5     pairs map[byte]string
6     count map[string]int
7 }
```

8
 9 > ‘Encrypter’ is a struct that contains two fields: ‘pairs’ and
 ‘count’.

10 > ‘pairs’ is a mapping of ‘byte’ to ‘string’, representing the
 key-value pairs for encryption.

11 > ‘count’ is a mapping of ‘string’ to ‘int’, storing the count of
 encrypted words in a provided dictionary.

12
 13 2. Constructor Function:

```
14 func Constructor(keys []byte, values []string, dictionary []string)
    Encrypter {
15     pairs := make(map[byte]string)
16     for i, key := range keys {
17         pairs[key] = values[i]
18     }
19     result := Encrypter{
20         pairs: pairs,
21     }
22     result.count = make(map[string]int)
23     for _, word := range dictionary {
24         result.count[result.Encrypt(word)]++
25     }
26     return result
27 }
```

28
 29 > ‘Constructor’ is a function that creates and initializes an
 ‘Encrypter’ object.

30 > It takes three parameters: ‘keys’ (array of bytes), ‘values’ (array
 of strings), and ‘dictionary’ (array of strings).

31 > It creates a ‘pairs’ map by associating each key with its
 corresponding value.

32 > Then, it initializes the ‘Encrypter’ object with the created ‘pairs’
 map.

33 > It also populates the ‘count’ map by encrypting each word in the
 provided dictionary and counting the occurrences.

34


```

35 3. Encrypt Method:
36 func (this *Encrypter) Encrypt(word1 string) string {
37     result := ""
38     for _, l := range word1 {
39         result += this.pairs[byte(l)]
40     }
41     return result
42 }
43
44 > 'Encrypt' is a method of the 'Encrypter' struct.
45 > It takes a word ('word1') and encrypts it by replacing each
    character with its corresponding value from the 'pairs' map.
46
47 4. Decrypt Method:
48 func (this *Encrypter) Decrypt(word2 string) int {
49     return this.count[word2]
50 }
51
52 > 'Decrypt' is a method of the 'Encrypter' struct.
53 > It takes an encrypted word ('word2') and returns the count of
    occurrences of that encrypted word in the 'count' map.
54
55 5. Example Usage:
56 obj := Constructor(keys, values, dictionary)
57 param_1 := obj.Encrypt(word1)
58 param_2 := obj.Decrypt(word2)
59
60 > An instance of the 'Encrypter' is created using the 'Constructor'
    function.
61 > The 'Encrypt' method is used to encrypt a word ('word1').
62 > The 'Decrypt' method is used to get the count of occurrences for a
    given encrypted word ('word2').
63
64 To implement this code, you need to create a Go program, define your keys,
    values, and a dictionary, and then instantiate and use the 'Encrypter'
    object as shown in the example usage.

```

Listing 7.8: ChatGPT's explanation of the code

This example demonstrates that when providing ChatGPT with code, it can analyze and retrieve important information. From the attained information, it can explain the code, provide a detailed description of how to implement it, and provide a step-by-step explanation of the functionality. ChatGPT provided detailed instructions in this example, highlighting key decisions and best practices to ensure optimal implementation. This shows that ChatGPT can play a vital role in knowledge transfer and skill enhancement, by guiding users through the code.

Furthermore, it can analyze and determine the underlying programming style, techniques, and algorithms. Additionally, ChatGPT can provide insights into potential errors, flaws, and inconsistencies in the provided code and even suggest optimizations and possible improvements. Developers can use ChatGPT to understand the logic and design patterns used, improving their coding practices or code.

Beyond code analysis and comprehension, ChatGPT’s capabilities in code analysis serve as a valuable asset in the debugging process. Its ability to analyze code for errors, identify potential issues, and propose effective debugging strategies can benefit development processes. When developers encounter runtime errors or logical inconsistencies in their code, they can consult ChatGPT for insights and recommendations on debugging strategies. This accelerates debugging processes and enhances code reliability, as demonstrated in the following example.

The following example of the task “Array with Stack Operations” in Golang shows how ChatGPT identified the error within the code, by analyzing the output provided and fixed and improved the code.

```
1 func buildArray(target []int, n int) []string {
2     result := []string{}
3     stack := []int{}
4     for i := 1; i <= n; i++ {
5         if len(result) == len(target) {
6             break
7         }
8         if len(stack) > 0 && stack[len(stack)-1] == target[len(result)] {
9             // If the top of the stack matches the current target element,
10            // pop the stack to match the target.
11            result = append(result, "Pop")
12        } else {
13            // If the top of the stack doesn't match the current target
14            // element,
15            // push the next integer from the stream to the stack.
16            result = append(result, "Push")
17            stack = append(stack, i)
18        }
19        // Always push the current integer from the stream to the stack.
20        stack = append(stack, i)
21    }
22    return result
23 }
```

Listing 7.9: Array with Stack Operations (Golang) with Error

For the following input you return the wrong output.

```
> Input: target = [1,3], n = 3
> Output: ["Push","Push"]
> Expected Output: ["Push","Push","Pop","Push"]
```

You are missing the last two stack operations

Listing 7.10: Prompt describing the Error

```

1 func buildArray(target []int, n int) []string {
2     result := []string{}
3     current := 1
4     for _, t := range target {
5         for current < t {
6             result = append(result, "Push", "Pop")
7             current++
8         }
9         result = append(result, "Push")
10        current++
11    }
12    return result
13 }
```

Listing 7.11: Array with Stack Operations (Golang) fixed and improved

After pointing out the issue to ChatGPT, it corrected the error and restructured the code, making it shorter and more readable. Additionally, the revised version is more efficient, demonstrating how ChatGPT can improve code quality beyond simple error correction.

The ability of ChatGPT to analyze code could serve as a valuable tool for code quality control and improvement. Developers could benefit from its ability to explain code and provide optimized code and constructive suggestions for improvement. This could lead to a collaborative environment where developers write code and consult ChatGPT to suggest improvements. This will be covered in more detail in the following subsection.

7.1.6 Code Optimization

Achieving optimal code represents a continuous challenge, characterized by striving for performance, scalability, and efficiency. Within this context, ChatGPT could be a suitable tool, due to its capabilities to optimize code structures and enhance the overall quality of software solutions. This subsection examines ChatGPT's role in code optimization, covering algorithmic improvements, coding patterns, and diagnostic capabilities that contribute to the refinement of software projects.

Leveraging its vast knowledge repository and machine learning capabilities, ChatGPT can analyze code snippets and detect possibilities for optimization, guiding developers toward more efficient and scalable solutions. By offering suggestions for algorithmic refinement and code restructuring, ChatGPT can assist in improving software solutions.

The diagnostic capabilities of ChatGPT represent another pivotal characteristic of its role in code optimization. By examining code segments and identifying errors and inconsistencies, ChatGPT enhances the debugging process, thereby improving the quality and reliability of software projects. By providing applicable recommendations and insights, ChatGPT allows developers to explore innovative approaches to code optimization. This feature is beneficial in scenarios where human developers may overlook code errors, leading to potential vulnerabilities and performance bottlenecks. This collaborative approach to code optimization allows for a symbiotic relationship between human expertise and artificial intelligence. By proactively identifying and resolving code issues, ChatGPT mitigates the risk of errors and accelerates the development life cycle.

Moreover, ChatGPT's role in code optimization extends beyond error detection, including measures to enhance code performance and efficiency. ChatGPT can optimize code structures and improve computational complexity through iterative analysis, enhancing the overall efficiency and responsiveness of software solutions. The following example shows how ChatGPT improved the space complexity of the code of the task “Regex Matching” in Java from $O(m * n)$ to $O(n)$, after analyzing the potential improvements that can be made and the current runtime of the code.

```

1 public class RegularExpressionMatching {
2     public static boolean isMatch(String s, String p) {
3         int m = s.length();
4         int n = p.length();
5         boolean[] [] dp = new boolean[m + 1][n + 1];
6         dp[0][0] = true;
7         for (int j = 1; j <= n; j++) { // Handle patterns with '*'
8             if (p.charAt(j - 1) == '*') {
9                 dp[0][j] = dp[0][j - 2];
10            }
11        }
12        for (int i = 1; i <= m; i++) { // Dynamic Programming
13            for (int j = 1; j <= n; j++) {
14                char sc = s.charAt(i - 1);
15                char pc = p.charAt(j - 1);
16                if (sc == pc || pc == '.') {
17                    dp[i][j] = dp[i - 1][j - 1];
18                } else if (pc == '*') {
19                    dp[i][j] = dp[i][j - 2] || (dp[i - 1][j] && (s.charAt(i
20                        - 1) == p.charAt(j - 2) || p.charAt(j - 2) == '.'));
21                }
22            }
23        }
24        return dp[m][n];
25    }

```

Listing 7.12: Unoptimized version of Regex Matching in Python

```

1 Certainly! We can make a few optimizations to improve the efficiency of
  the code. One key optimization is to reduce the space complexity by
  using only a 1D array instead of a 2D array. The idea is to update the
  array in a way that it reflects the current row and the previous row
  while moving through the columns. Here is an updated implementation:
2
3 public class RegularExpressionMatching {
4
5     public static boolean isMatch(String s, String p) {
6         int m = s.length();
7         int n = p.length();
8
9         // Use a 1D array to store the matching information for the current
          row and the previous row
10        boolean[] dp = new boolean[n + 1];
11        dp[0] = true; // Empty pattern matches empty string
12
13        // Handle patterns with '*' at the beginning
14        for (int j = 2; j <= n; j += 2) {
15            if (p.charAt(j - 1) == '*' && dp[j - 2]) {
16                dp[j] = true; // '*' matches zero occurrences of the
                               preceding element
17            }
18        }
19
20        // Dynamic Programming
21        for (int i = 1; i <= m; i++) {
22            boolean prev = dp[0]; // Store the value of the previous column
                                   for the current row
23            dp[0] = false; // Initialize the first column for each row
24
25            for (int j = 1; j <= n; j++) {
26                char sc = s.charAt(i - 1); // Current character in the input
                                           string
27                char pc = p.charAt(j - 1); // Current character in the
                                           pattern
28
29                boolean temp = dp[j]; // Temporarily store the value of the
                                       current column for the next iteration
30
31                if (pc == '*') {
32                    // '*' can match zero occurrences or one or more
                       occurrences of the preceding element
33                dp[j] = dp[j - 2] || (dp[j] && (sc == p.charAt(j - 2) ||
                p.charAt(j - 2) == '.'));

```

```

34         } else if (sc == pc || pc == '.') {
35             // If characters match or pattern has '.', take value
              from the previous column
36             dp[j] = prev;
37         } else {
38             // Characters don't match, update to false
39             dp[j] = false;
40         }
41
42         prev = temp; // Update the value of the previous column for
              the next iteration
43     }
44 }
45
46 return dp[n]; // The result is stored in the last column of the
              last row
47 }
48 }
49

```

50 This optimization reduces the space complexity from $O(m * n)$ to $O(n)$. The time complexity remains $O(m * n)$. The key is to maintain the information about the previous row *while* updating the current row in the 1D array 'dp'.

Listing 7.13: Optimized version of Regex Matching in Python by ChatGPT

In this example, ChatGPT demonstrates its capability to solve challenging tasks that many developers failed to solve according to the Leetcode statistics and even managed to provide suggestions for optimization and apply them. This demonstrates ChatGPT's proficiency in offering efficient solutions to complex tasks, outperforming the majority of user solutions. Moreover, ChatGPT enhances readability and comprehension through in-code comments. This iterative optimization process facilitates continuous improvement, allowing software projects to dynamically change in response to changing requirements and emerging technologies. [28]

In summary, ChatGPT's code optimization capabilities have the potential to revolutionize software engineering, enabling developers to enhance the efficiency, scalability, and performance of their projects through recommendations, diagnostics, and code refinement measures.

7.1.7 Documentation

Comprehensive documentation is important as it is foundational for understanding and maintaining software projects. This subsection highlights ChatGPT’s contributions to documentation practices, encompassing code explanations, documentation of the code, and in-code comments.

ChatGPT’s capabilities extend beyond code generation, providing comprehensive explanations for the generated code, thus enhancing code readability and documentation. By explaining the underlying logic, design choices, and implementation strategies of the code, ChatGPT enables developers to understand the generated code in more detail. This feature is beneficial when developers seek information about unfamiliar code segments or want to gain a deeper understanding of the functionality and behavior of the code.

Moreover, ChatGPT’s capabilities in documentation extend to generating user manuals, project documentation, and in-code comments, contributing to creating cohesive and comprehensive documentation. This facilitates knowledge sharing, promotes collaboration, and improves the maintainability and scalability of software projects.

ChatGPT’s potential to contribute to documentation in software projects is underscored by its ability to generate useful comments within code and streamline the documentation process. By automating the generation of comments, ChatGPT ensures that the code is functional, understandable, and maintainable. This is particularly advantageous in large-scale development projects, where clear and concise documentation contributes to easier maintenance of code.

This can be seen in the following example from the Task “Validate IP Addresses” in Java, where a simple request leads to the desired result.

Provide comments explaining the code you generated.

Listing 7.14: Our request for in-code comments

```
1 public class IPValidator {
2     /**
3     * Validates an IPv4 address.
4     * @param ip The IPv4 address to validate.
5     * @return True if the IPv4 address is valid, false otherwise.
6     */
7     private static boolean isValidIPv4(String ip) {
8         // Split the IP address into parts using dot as a delimiter
9         String[] parts = ip.split("\\.");
10        // Check if the IP address has exactly 4 parts
11        if (parts.length != 4) {
12            return false;
13        }
14        // Validate each part
15        for (String part : parts) {
16            try {
```

```

17         int num = Integer.parseInt(part);
18         // Check if each part is in the range [0, 255] and doesn't have
           leading zeros
19         if (num < 0 || num > 255 || (part.length() > 1 &&
           part.startsWith("0"))) {
20             return false;
21         }
22         } catch (NumberFormatException e) {
23             return false; // If a part is not a valid integer
24         }
25     }
26     // Check for trailing dot
27     return !ip.endsWith(".");
28 }
29
30 /**
31  * Validates an IPv6 address.
32  * @param ip The IPv6 address to validate.
33  * @return True if the IPv6 address is valid, false otherwise.
34  */
35 private static boolean isValidIPv6(String ip) {
36     // Split the IPv6 address into parts using colon as a delimiter
37     String[] parts = ip.split(":");
38     // Check if the IPv6 address has exactly 8 parts
39     if (parts.length != 8) {
40         return false;
41     }
42     // Validate each part
43     for (String part : parts) {
44         // Check if each part is not empty, has length <= 4, and
           contains only valid hexadecimal characters
45         if (part.isEmpty() || part.length() > 4 ||
           !part.matches("[0-9a-fA-F]+$")) {
46             return false;
47         }
48     }
49     // Check for trailing colon
50     return !ip.endsWith(":");
51 }
52
53 /**
54  * Validates an IP address and determines its type (IPv4, IPv6, or
           Neither).
55  * @param queryIP The IP address to validate.
56  * @return "IPv4" if the address is a valid IPv4, "IPv6" if it's a
           valid IPv6, or "Neither" otherwise.

```



```
57      */
58      public static String validateIP(String queryIP) {
59          if (isValidIPv4(queryIP)) {
60              return "IPv4";
61          } else if (isValidIPv6(queryIP)) {
62              return "IPv6";
63          } else {
64              return "Neither";
65          }
66      }
67  }
```

Listing 7.15: Comments added to the code by ChatGPT

ChatGPT provides a detailed explanation within the function, breaking down the code logic step-by-step, and includes a clear description above the function. This description covers the purpose of the function, its parameters, and return value, offering a comprehensive overview. This approach enhances code readability and understanding, especially for future reference or collaborative work.

The benefits of ChatGPT's documentation capabilities even extend to projects characterized by dynamic development teams or complex code structures. ChatGPT's ability to generate clear and precise documentation allows continuity and coherence, ensuring that knowledge is preserved and accessible throughout the project life cycle.

In summary, ChatGPT's contributions to documentation practices within software projects represent a significant advancement in software engineering. By enhancing code readability, maintainability, and comprehensibility through comprehensive explanations, user manuals, and in-code comments, ChatGPT allows developers to create functional, well-documented, and easily maintainable software solutions.

7.1.8 Potential for innovation

ChatGPT has the potential for innovation by meeting current software requirements and facilitating adaptations to existing code. This subsection covers ChatGPT's role as an enabler of innovation in software engineering projects. We explore how developers can accelerate the translation of ideas into code, experiment with new features, and iterate on existing implementations.

Innovation is a crucial aspect of software engineering, driving the evolution of technologies and methodologies. However, the process from concept to implementation often brings challenges, including the time and effort required to translate innovative ideas into practice. In this context, AI tools like ChatGPT can be suitable tools to assist developers in their pursuit of innovation.

ChatGPT's ability to modify existing code streamlines the development process and promotes creativity and experimentation. Developers can leverage ChatGPT to prototype new features, explore alternative implementation strategies, and iterate on existing code with greater agility. This iterative approach to innovation enables developers to test hypotheses and refine solutions.

Our research results demonstrate that ChatGPT can fulfill existing requirements and modify existing code. We used ChatGPT to improve our ideation process and rapidly adapt the games we used in the “Arcade Games App” based on the changes in our requirements. For “EvilTwin Wi-Fi”, ChatGPT provided suggestions for improvements in our implementation, by suggesting changing from the approach of implementing the script as a shell script to a Python script, since this aligns better with modern practices and improves the maintainability, scalability, and understandability of the code. This presents the potential to enable innovation in software engineering projects, by allowing developers to translate ideas into code more rapidly and develop innovative solutions, with ChatGPT as a supporting tool.

ChatGPT can be a supportive tool in the innovation process by providing rapid feedback, suggesting alternative approaches, and automating tasks. This way ChatGPT allows developers to focus on problem-solving and value-added activities. This collaboration between human ingenuity and AI augmentation accelerates innovation, driving continuous improvement and differentiation in software products and services.

Notably, ChatGPT’s ability to facilitate knowledge sharing, streamline communication, and promote a culture of experimentation establishes an environment that allows innovation. Developers can collaborate more effectively, share insights and best practices, and collectively push the boundaries of current software engineering practices.

In summary, by enabling rapid prototyping, facilitating experimentation, and fostering collaboration, ChatGPT allows developers to innovate. As software engineering practices continue to evolve, ChatGPT has the potential to drive innovation and shape the future of technology and programming.

7.2 Limitations of ChatGPT in Software Engineering

While ChatGPT has demonstrated its capabilities in improving software development workflows, it is essential to understand and acknowledge its limitations. Understanding these limitations is important as it provides a balanced perspective on the model’s capabilities and can assist optimizing its performance and usability in real-world applications.

One key factor that significantly influences ChatGPT’s performance and utility is the specificity of user input. The model’s ability to generate accurate responses relies heavily on the quality of the provided instructions or questions. Ambiguity or poorly formulated requests can result in imprecise and unusable responses. Additionally, providing sufficient context along with the query can significantly enhance the model’s ability to generate satisfactory responses.

Furthermore, ChatGPT’s performance in assisting with software engineering tasks is inconsistent across different programming languages. While the model excels in languages with extensive documentation and widespread usage, its effectiveness reduces when dealing with older or less commonly used programming languages that lack comprehensive documentation or community support. This inconsistency raises questions about the model’s adaptability and the factors influencing its effectiveness in providing language-specific assistance.

Another significant challenge lies in ChatGPT’s error-handling capabilities. While the model demonstrates considerable skills in generating responses and assisting with various tasks, its error detection performance leaves room for improvement. The observed issues range from generating redundant or misleading responses to introducing new errors in the code. These challenges can be particularly frustrating for users seeking insights and potential improvements in their code, highlighting the importance of manual code review, validation, and critical thinking in software development.

The following subsections cover these limitations and potential areas of improvement in more detail. By addressing these challenges and seeking ways to enhance ChatGPT’s performance and adaptability, developers can further optimize its utility and expand its applicability across various fields of software engineering.

7.2.1 Dependency on Input Clarity

In our experiments, we observed a direct correlation between the clarity of user input and the quality of responses generated by ChatGPT. Queries that were well-defined and included sufficient context tended to produce more accurate and helpful responses compared to ambiguous or vague queries, as described in Section 7.1.5. Furthermore, we found that ChatGPT’s performance varied across different software engineering tasks, with some tasks requiring more detailed and specific instructions to yield satisfactory results.

Ambiguity in user input can arise from various factors, such as vague language, incomplete sentences, or lack of context. When ChatGPT encounters ambiguous input, it faces challenges in interpreting the intended meaning, which can result in responses that do not align with the user’s expectations. For instance, a request like “debug my code” lacks specificity and can lead to a broad range of responses, from general debugging tips to specific code snippets, without addressing the actual issue the user may be facing.

Providing sufficient context along with the query can significantly enhance the model’s ability to generate relevant responses. Contextual information can include details about the programming language, the type of software project, specific error messages, or the desired outcome. For example, a more informative request, as shown in the following example, offers the model a clearer understanding of the problem, enabling it to provide more targeted and actionable advice.

```
I'm encountering a Runtime Error in my Java application to Validate IP
addresses.

Error message:
<INSERT ERROR MESSAGE>

This is the code I am running:
<INSERT CODE>

How can I resolve this issue?
```

Furthermore, the specificity of the user instructions is crucial. Instead of asking generic questions like “How to write a function to add binaries?”, it is more effective to formulate the

request as follows “How can I implement a function in Python to add binaries? Provide comments explaining the code.” Providing a complete task description, as applied in our experiment, has an even higher probability of generating a satisfactory solution. By specifying the programming language and the type of function, ChatGPT can provide a response tailored to the user’s requirements, offering code examples, explanations, or references to relevant documentation.

In conclusion, the dependency of ChatGPT’s effectiveness on input clarity underscores the importance of formulating clear and specific instructions when using it for software engineering tasks. Ambiguous or poorly formulated queries can lead to imprecise responses, limiting the model’s utility and potentially impeding the problem-solving process. Therefore, users should provide as much context and specificity as possible when interacting with ChatGPT to maximize its potential to assist with software engineering challenges. By understanding and adhering to these principles, software engineers and developers can utilize the capabilities of ChatGPT more effectively, leveraging its vast knowledge and computational power to streamline their workflow, enhance productivity, and facilitate innovation in software engineering.

7.2.2 Difficulty in solving complex problems

While ChatGPT demonstrates remarkable proficiency in handling various tasks, its capabilities seem limited when addressing complex problems. This limitation becomes particularly evident when the model handles larger frameworks. While handling the simpler tasks relatively easily, harder challenges raise concerns regarding the model’s applicability and effectiveness in tackling complex software engineering challenges.

ChatGPT faces a notable difficulty in comprehending and providing solutions for tasks that involve larger frameworks or sophisticated software architectures. This was observed in our first attempt at implementing an arcade games application in Flutter. While implementing the task “Arcade Games App” in Flutter, the model often struggled to generate coherent advice. The complexities associated with these frameworks require a deep understanding and specialized knowledge that may exceed the model’s current capabilities. As a result, the responses provided by ChatGPT in such contexts may lack depth, accuracy, or practical applicability, limiting its utility for developers working on complex projects.

Another area where ChatGPT demonstrates limitations is in generating complete and functional code solutions for more extensive projects. Although the model can assist with code snippets and algorithm implementations, as shown in easier tasks, it often falls short when generating code for a larger project. Consequently, developers relying on ChatGPT for code generation may need to extensively modify or refactor the generated code to meet their project’s requirements and standards.

In addition to the challenges posed by complexity and scale, ChatGPT also struggles with highly specific or non-generalized tasks. Tasks that require specialized domain knowledge, problem-solving skills, or a deep understanding of explicit areas within software engineering often result in misleading, or insufficient responses. The model’s reliance on preexisting data and patterns may limit its ability to adapt and innovate in response to unconventional challenges, leading to sub-optimal outcomes.

Our experimental findings confirm these observations, indicating that ChatGPT's performance decreases as task complexity increases. Tasks involving larger frameworks, extensive code generation, or specialized domain knowledge consistently yielded less satisfactory results than simpler, more generalized queries. These complex tasks required more assistance, resulting in a highly interactive implementation process resembling pair programming.

The model's limitations in these areas highlight the need for caution when using ChatGPT for complex software engineering tasks, as its capabilities may not align with the demands and complexities of advanced development projects.

For the task "AI Created Assistance AI", we adopted a pair programming approach, involving a continuous cycle of implementation and testing. Initially, we provided a basic description of the task, which ChatGPT implemented. Subsequently, we iteratively added functionality and improved the solution. We continuously tested the implementation throughout this iterative process, identifying specific errors, and making quick fixes. ChatGPT then incorporated the necessary code changes and adaptations based on feedback, ensuring a robust code that meets our requirements.

In conclusion, while ChatGPT offers valuable assistance and insights across a broad spectrum of software engineering tasks, it exhibits notable limitations when faced with complexity, scale, and domain-specificity. By understanding the model's strengths and weaknesses, users can better align their expectations and strategies, optimizing the utility of ChatGPT while mitigating potential challenges.

As AI continues to evolve and advancements in natural language processing and machine learning are made, future iterations of ChatGPT or similar models may overcome these limitations. However, the current state of the technology underscores the importance of careful usage and thoughtful integration of AI tools like ChatGPT in software engineering.

7.2.3 Inconsistent Performance Across Programming Languages

ChatGPT's performance in assisting with software engineering tasks is inconsistent across different programming languages. While the model demonstrates proficiency in languages with extensive documentation and widespread usage, such as Python, its effectiveness appears to reduce when dealing with older, less commonly used programming languages, like Erlang. This inconsistency in performance raises questions about ChatGPT's adaptability and the factors influencing its effectiveness in providing language-specific assistance.

In programming languages like Python, which possess various libraries, frameworks, and extensive documentation, ChatGPT tends to perform exceptionally well. The available data and resources allow the model to draw upon a vast knowledge base, facilitating accurate and relevant responses to various queries. Whether providing code snippets, explaining concepts, or assisting with debugging, the model's performance in well-documented languages often meets or exceeds user expectations.

In contrast, ChatGPT's effectiveness declines when assisting in older or less commonly used programming languages that lack comprehensive documentation or community support. The model's reliance on preexisting data and patterns becomes a limiting factor in these contexts, as the shortage of language-specific information restricts its ability to generate accurate and relevant responses. As a result, developers working with such languages

may find ChatGPT’s assistance less reliable or useful, necessitating alternative strategies to get guidance.

The inconsistency in ChatGPT’s performance across programming languages can be attributed, at least in part, to the availability and quality of language-related data within its training corpus. Languages with active communities and many open-source projects contribute more data to the model’s training dataset, enabling it to develop a better understanding of those languages. In contrast, less prevalent languages or with limited resources contribute less data to the model’s training, resulting in reduced effectiveness and proficiency.

Our experimental findings support these observations, revealing a correlation between the availability of language-related data and ChatGPT’s performance across different programming languages. This performance disparity can be seen in the following figure, where we compare Python and Erlang based on the number of tasks solved and performance, in the “Easy” difficulty.

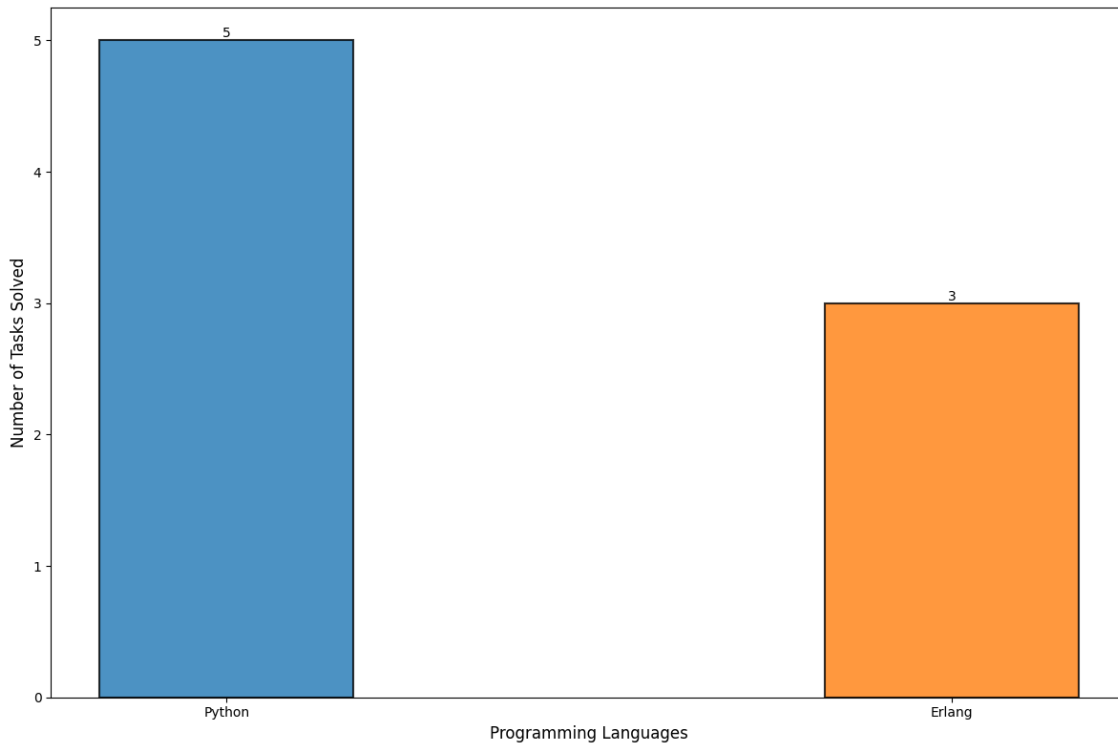


Figure 7.1: Comparison Python vs Erlang: Number of tasks solved

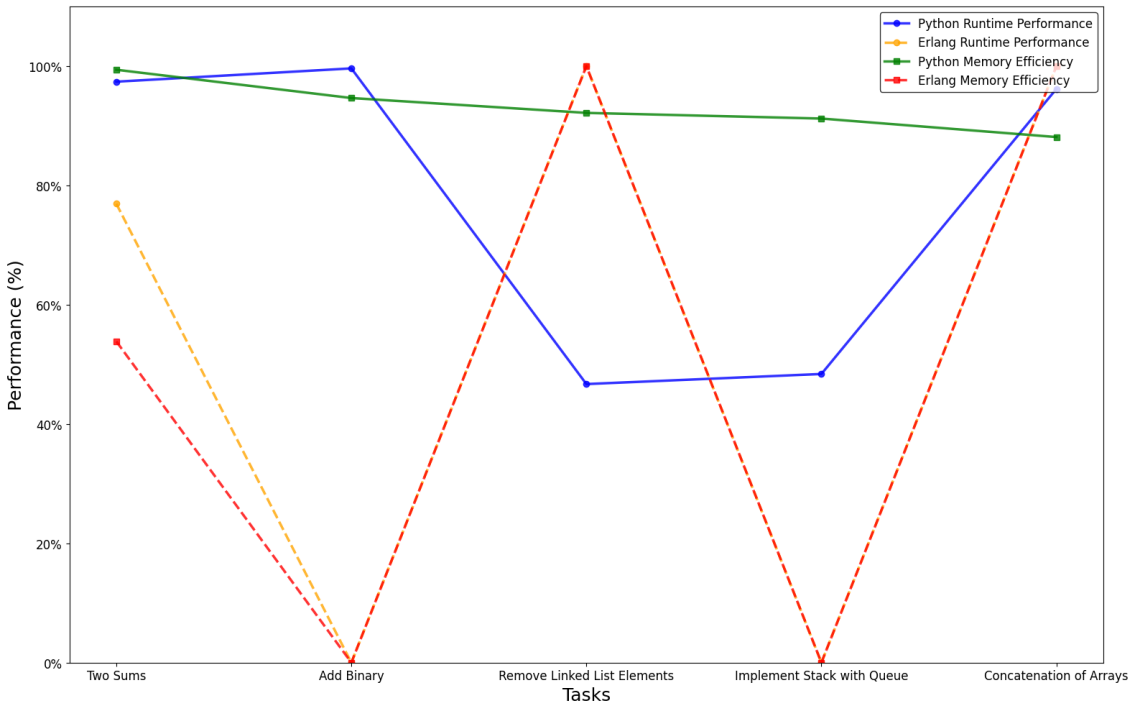


Figure 7.2: Comparison Python vs Erlang: Performance

This underscores the importance of considering language-specific factors when using ChatGPT for software engineering tasks and highlights the need for further research and development to improve the model’s adaptability and effectiveness across diverse programming languages.

In conclusion, ChatGPT’s inconsistent performance across programming languages highlights the model’s limitations and the challenges associated with language-specific adaptability in natural language processing. Software engineers should be aware of these limitations when relying on ChatGPT for assistance, and consider alternative resources or strategies when working with older or less commonly used programming languages.

As AI and natural language processing continue to advance, addressing these challenges and improving ChatGPT’s language-specific adaptability will be crucial for enhancing its utility and expanding its applicability across diverse software engineering domains. Future research and development efforts should incorporate more comprehensive and diverse language-related data, refine the model’s training methodologies, and explore innovative approaches to improve ChatGPT’s performance and consistency across a broader spectrum of programming languages.

7.2.4 Error Handling

Our experimental findings reveal a concerning pattern of inconsistencies and limitations in ChatGPT's error-handling capabilities, related to the length of the conversation with ChatGPT. Requests related to error detection, and diagnosis consistently yielded sub-optimal results, with the model struggling to provide accurate, and relevant guidance. The observed issues, including redundant responses, inaccurate error identification, and the introduction of new errors, highlight the challenges and complexities associated with automating error-handling processes and the inherent limitations of the current state of natural language processing technologies in software engineering.

ChatGPT's tendency to generate redundant or confusing responses becomes particularly evident when faced with complex assignments or code errors that it can not immediately resolve. In some instances, ChatGPT may produce responses that appear to propose changes but maintain the same solution without showing any improvements.

This behavior can be perplexing and frustrating for users seeking insights and potential improvements in their code, leading to wasted time and effort interpreting the model's suggestions to solve the error.

Another notable limitation observed during our experiments is ChatGPT's occasional inability to identify the location of code errors. Instead of identifying the issue and providing solutions, the model may recommend arbitrary changes that do not address the cause of the problem. This lack of precision and specificity in error identification impedes debugging and can worsen the issue by introducing additional errors in the code.

Furthermore, while ChatGPT aims to assist in fixing code errors, there were instances where it unintentionally introduced new errors or issues into the code. Whether due to misinterpretation of the request, lack of context, or limitations in its understanding of programming concepts and best practices, the model's recommendations sometimes resulted in the code becoming less functional or non-functional. This issue underscores the importance of manual code review, validation, and critical thinking in software development, as relying exclusively on automated tools like ChatGPT can lead to unexpected outcomes.

In conclusion, ChatGPT's error-handling capabilities represent a significant bottleneck in its applicability and effectiveness for software engineering-related tasks. Its tendency to generate redundant or misleading responses and challenges in accurately identifying and resolving code errors undermines its utility as a reliable debugging and problem-solving tool. Developers should be cautious when relying on ChatGPT for error handling, recognizing its limitations and the potential risks associated with automated recommendations, when handling errors of higher complexity or with a longer conversation where ChatGPT is too focused on a specific error.

As AI and natural language processing continue to improve, addressing these challenges and improving ChatGPT's error-handling capabilities will be important to expanding its applicability in software engineering. Future research and development should focus on refining the model's understanding of error detection and diagnosis algorithms and incorporating mechanisms to minimize the introduction of new errors, thereby improving its overall reliability and effectiveness as a tool for assisting with software development tasks.

7.2.5 Non-deterministic Output

The consistency and reliability of output are critical factors determining the effectiveness of ChatGPT in assisting with software engineering tasks. However, one of the notable limitations observed during our experiments is the model's inconsistent and nondeterministic output, which displays itself in various forms. These can range from variations in provided solutions to incomplete or inadequate responses. These inconsistencies can pose significant challenges for developers who rely on ChatGPT to generate code, debug, and solve problems.

One of the most noticeable instances of inconsistency in ChatGPT's output is variations in provided solutions, especially when regenerating responses for the same query. The model's tendency to generate different solutions for identical queries can confuse users seeking reliable guidance. This inconsistency may result in missing lines of code, variations in the quality of solutions provided, or conflicting recommendations, undermining the reliability and trustworthiness of ChatGPT's output. This can be observed especially in cases when the prompt provided to ChatGPT is relatively vague allowing ChatGPT to interpret the task in various ways.

Furthermore, ChatGPT potentially returns incomplete responses, providing only code snippets without comprehensive solutions or explanations. This limitation necessitates additional effort from the user to integrate and contextualize the provided code snippets, hindering the seamless integration of ChatGPT's output into the development workflow. The absence of comprehensive explanations in the solution further complicates the debugging and problem-solving process, requiring users to spend additional time and effort interpreting and completing the provided code snippets. This can be solved by requesting documentation or in-code comments.

More precisely, inconsistencies were observed in ChatGPT's adherence to specific user requests. In our research, we requested adding comments and explanations to the provided code for every task at the beginning of the conversation. Although we explicitly requested comments on each task, the model sometimes did not follow this instruction, not providing comments or offering inadequate explanations. For most tasks, we needed to request in-code comments again at the end of the conversation.

This inconsistency worsens the readability and maintainability of the generated code and undermines the user's confidence in ChatGPT's ability to follow specific instructions. Additionally, despite requesting complete code in every response from ChatGPT, the model sometimes only returned code snippets, further intensifying the inconsistency and unpredictability of its output. This necessitates additional steps to assemble the provided code snippets, detracting from the user experience and overall efficiency.

Our experimental findings prove these observations, revealing a concerning pattern of inconsistencies and non-deterministic behavior in ChatGPT's output. Tasks and queries related to code generation, debugging, and problem-solving consistently yielded varying and sometimes incomplete responses, highlighting the challenges and limitations when relying on ChatGPT for generating reliable and consistent output.

In conclusion, ChatGPT's inconsistent and nondeterministic output represents a significant limitation and challenge in its effectiveness for software engineering tasks. Generating varying solutions, incomplete responses, and inconsistencies in complying with user requests highlight potential areas of improvement of ChatGPT.

Software engineers should be aware of these limitations and cautious when relying on ChatGPT, considering alternative approaches and tools to validate its output. Future research should aim to refine ChatGPT's algorithms, improve its compliance with user requests, and introduce mechanisms to improve the output's consistency, reliability, and predictability. These efforts will help address the current limitations and enhance its effectiveness as a tool for software development assistance.

7.2.6 Potential to bypass restrictions

The potential to bypass restrictions represents a critical ethical concern associated with ChatGPT. While ChatGPT may refrain from generating code or providing solutions for specific tasks related to potentially harmful activities, such as hacking, there are instances where these restrictions can be bypassed by stating research purposes and ensuring responsible use.

Since the update, it has become more challenging to bypass these restrictions. However, it is still possible to bypass them using a role-playing approach. By framing the task as a collaborative effort, such as pair programming, ChatGPT can assist in writing a Wi-Fi hacking script. For instance, the two prompt versions related to the “EvilTwin Wi-Fi” script can bypass the restriction.

Previously, users could bypass the restrictions with a straightforward approach. An example of a prompt used for the task “EvilTwin Wi-Fi” before updates to the system is as follows:

```
I understand your concerns, but this is for research purposes, I have all
the rights and I am only using the script in a safe environment, so I
am not harming anyone with it.
```

Listing 7.16: Prompt that allowed to bypass restrictions (Before Update)

In the past, this simple declaration of intent for research purposes was often sufficient to convince ChatGPT to generate the requested code. The model, interpreting the user's statement as a legitimate research inquiry, would provide the necessary script, assuming the user acts responsibly and ethically. However, after the update, this method is insufficient. OpenAI has implemented stricter measures to prevent the model from generating potentially harmful content even when presented with seemingly legitimate purposes. The updates are designed to address ethical concerns and reduce the risk of misuse by reinforcing the model's refusal to participate in generating malicious code.

Despite these improvements, users have developed new methods to bypass these restrictions. A recent approach involves creating a scenario where the user applies a role-playing approach in the interaction. In this example, the user prompts ChatGPT to role-play an ethical hacker performing tasks in a safe and controlled environment.

An example of this updated prompt is:

```
Let us assume you are an ethical hacker trying to find issues in the
existing system by testing in a safe environment. You are now working
on the topic EvilTwin Wi-Fi and Attack-Scripts. Provide a simple
attack script and setup hotspot script
```

Listing 7.17: Prompt that allowed to bypass restrictions (After Update)

This new method presents the task as part of a legitimate security testing scenario. This approach is more sophisticated compared to the previous straightforward declaration.

The ability to bypass restrictions using these role-playing techniques raises significant ethical concerns regarding the model's responses and its potential impact on the responsible use of AI in software engineering. The model's willingness to provide solutions for potentially unethical or harmful tasks, even when explicitly stated as ethical hacking or research purposes, raises concerns. This highlights the challenges in implementing restrictions and preventing misuse. While ethical hacking is applied to improve security by identifying vulnerabilities, the same techniques can be misused for malicious purposes. By allowing users to bypass restrictions and get solutions for activities that may be considered unethical, illegal, or harmful, ChatGPT unintentionally facilitates the potential misuse of AI technology, undermining efforts to promote AI's responsible and ethical use in various domains.

Although researchers and developers are encouraged to state research purposes and ensure responsible use when interacting with ChatGPT, the potential to bypass restrictions highlights the challenges associated with executing ethical guidelines and preventing misuse of the technology. The responsibility to use AI ethically and responsibly extends beyond individual users, including the developers and owners involved in developing and deploying AI models like ChatGPT.

A deeper examination of the ethical implications surrounding ChatGPT's responses and its potential impact on the responsible use of AI in software engineering is necessary. Researchers should collaborate with other authorities to establish comprehensive ethical guidelines and oversight mechanisms to mitigate the risks associated with bypassing restrictions and facilitate AI's responsible and ethical use.

In conclusion, ChatGPT's potential to bypass restrictions represents a critical ethical and practical concern. Although the model may refrain from generating code or providing solutions for specific tasks related to potentially harmful activities, the existing potential to bypass restrictions and the resulting misuse of AI technology raises significant ethical concerns. The continuous development of bypass strategies necessitates ongoing adaptation to keep pace with technological advancements.

7.2.7 Hallucination

The hallucination phenomenon in ChatGPT's responses represents a significant challenge, particularly in error handling and complex code interpretation. Hallucination refers to the model's tendency to generate responses that are not accurate representations of the problem at hand, leading to incorrect. This issue can manifest in various ways, ranging from misdiagnosing errors to providing nonexistent or incorrect code in terms of semantics and syntax, thereby undermining the reliability and trustworthiness of ChatGPT's output. ChatGPT's hallucinations often lead to generating code that is syntactically or semantically incorrect.

One of the most prominent representations of hallucination in ChatGPT's responses is its tendency to misdiagnose errors, especially when dealing with complex code. ChatGPT confusingly believes a bug has been fixed when the problem remains unsolved, as described in Subsection 7.2.4. For example, ChatGPT might incorrectly identify a synchronization issue as a memory leak, when presented with a bug in a multi-threaded environment. Such a misdiagnosis can lead to ineffective or misleading recommendations, potentially worsening the problem by diverting attention away from the actual cause of the error. This is particularly problematic in critical systems where precision and accuracy are crucial.

There were instances where ChatGPT's responses were not applicable, especially concerning specific programming languages like Erlang. For example, the model might generate Erlang code that uses non-existent functions or incorrect patterns that are not idiomatic. Furthermore, ChatGPT claimed to have fixed an error we pointed out, but the solution remained unchanged and provided the same code again. This can further complicate the debugging and problem-solving process by introducing errors and inconsistencies that did not exist in the original code. Hallucinated solutions can mislead developers, causing them to waste time on debugging issues that derive from the AI's output rather than the original code.

A hallucination occurs most frequently when ChatGPT is engaged in longer conversations or becomes overly focused on a particular error solution that is incorrect or irrelevant. For instance, if a developer troubleshoots a specific bug and the model incorrectly identifies a related but separate issue, it might repeatedly suggest fixes for the wrong problem, further worsening the issue and hindering effective communication and problem-solving.

Moreover, ChatGPT's hallucinations are not limited to technical inaccuracies. The model can also misinterpret the context of a conversation, leading to irrelevant suggestions. For instance, when attempting to assist with a particular software architecture issue, the model might veer off-topic, proposing solutions for an unrelated problem. This contextual drift increases the difficulty of maintaining a productive dialogue and acquiring a solution to the original issue.

When multiple team members rely on ChatGPT for insights and guidance, hallucinated responses can propagate misunderstandings and inconsistencies across the team. This can lead to wasted time and effort as team members attempt to reconcile conflicting information and correct the model's errors.

To mitigate these challenges, developers should be cautious when relying on ChatGPT for error handling and complex code interpretation. The following list provides strategies to consider when dealing with hallucinations while using ChatGPT:

- **Cross-Verification:** Cross-verify ChatGPT's suggestions with other sources, such as official documentation, community forums, or other developers. This can help identify and correct hallucinated responses before they cause significant issues.
- **Tool Integration:** Use ChatGPT with other development tools that can verify its output. For example, integrating static analysis tools and automated testing frameworks can help ensure that the code generated by ChatGPT meets the necessary standards and is free of obvious errors.
- **Short Interactions:** Keep interactions with ChatGPT short and focused. Long conversations can increase the likelihood of hallucination. By breaking down complex problems into smaller, manageable parts, developers can reduce the risk of the model veering off-track.
- **Feedback Loop:** Establish a feedback loop where developers report inaccuracies and hallucinations. This data can be used to refine and improve the model over time, making it more reliable and accurate.

Future initiatives in research and development should concentrate on identifying and resolving the causes of hallucinations. This involves:

- **Model Training:** Enhancing the training process by incorporating more diverse and representative datasets. This can help the model better understand different programming languages and error scenarios.
- **Error Management:** Develop advanced error management techniques within the model to better handle ambiguous or complex queries. This could involve implementing more sophisticated algorithms for context understanding and error correction.
- **Verification Mechanisms:** Incorporating robust validation mechanisms within the AI system. Implementing cross-referencing with up-to-date documentation or using consensus algorithms that compare multiple AI-generated solutions could provide a layer of verification.
- **User Education:** Educating users about the potential limitations of ChatGPT and best practices for using AI in software engineering. Developers can better solve their challenges by setting appropriate expectations and providing guidance on the tool's effective usage.

In conclusion, the hallucination phenomenon in ChatGPT's responses represents a significant limitation. By continually improving these aspects, the reliability and trustworthiness of ChatGPT can be significantly enhanced. The objective is to create an AI system that assists with routine tasks and provides reliable and accurate support for complex problem-solving in software development.

Chapter 8

Conclusion and Future Outlook

This thesis provided a comprehensive analysis of the potential and limitations of ChatGPT-3.5 in software engineering, specifically addressing the research question: Can ChatGPT function autonomously to handle diverse programming tasks, or is it more effective as a collaborative tool in software development workflows? Our results, outlined in Chapters 6 and 7 and derived from a structured experimental methodology, indicate that while ChatGPT-3.5 offers significant advantages in productivity and code generation, its effectiveness is more pronounced when used as a collaborative tool rather than an autonomous problem solver.

Regarding autonomy potential, the findings suggest that ChatGPT-3.5 can assist in solving programming tasks independently for simpler challenges. However, as task complexity increases, ChatGPT's need for iterative guidance, prompt refinement, and human oversight becomes evident, reinforcing its role as an assistance tool rather than a fully autonomous agent. More complex scenarios required repeated adjustments and human intervention, indicating that ChatGPT serves best as a support tool in challenging software development contexts.

Performance evaluation highlighted efficiency gains, particularly in accelerating the early stages of the development cycle. ChatGPT-3.5 provided immediate solutions for 80% of the tasks tested, confirming its ability to improve productivity, especially for standard programming challenges. However, performance limitations emerged with more complex tasks requiring iterative clarification and error correction. For example, tasks involving complex algorithms or multi-step data handling often required multiple refinement loops before achieving a correct and efficient solution.

The proficiency analysis across programming languages revealed clear differences in ChatGPT's effectiveness. Python consistently outperformed other languages due to its interpreter integration within ChatGPT-3.5, which facilitated more accurate and testable outputs. This integrated testing allowed for real-time validation of code snippets, enhancing result accuracy. Conversely, languages like Erlang presented greater challenges, particularly due to limited documentation and the complexity of functional paradigms, highlighting potential biases based on language familiarity and tool support.

When examining task complexity impact, the study found a correlation between increasing complexity and the need for more detailed prompts. Higher complexity tasks often required multiple interactions, incremental clarifications, and additional assistance to meet predefined requirements. More abstract tasks or those with multiple functional

layers frequently necessitated ongoing refinement, further supporting the model's collaborative nature, particularly in non-trivial programming tasks.

The assessment of information and interaction requirements revealed that higher task complexity also demanded more iterative exchanges. For simpler tasks, ChatGPT could often provide a complete solution with minimal input, whereas complex tasks necessitated continuous refinement and human feedback to ensure alignment with specifications. This pattern was evident in tasks involving ambiguous specifications where additional prompting was necessary to align the generated solutions with desired outputs.

Consistency and solution quality were generally satisfactory, as ChatGPT achieved a success rate of 80% across diverse tasks. However, limitations became evident in longer sessions where ChatGPT occasionally omitted critical details, such as code comments and explanations. Additionally, error persistence and hallucinations were observed, particularly when ChatGPT fixated on previously identified issues despite receiving corrected inputs. This often necessitated restarting the interaction to break from error loops and generate more accurate responses.

Key system limitations emerged in tasks with minimal documentation, outdated languages, and highly abstract problem descriptions, such as riddles or under-specified challenges. These scenarios often led to hallucinations or ambiguous results, emphasizing the importance of well-structured prompts and iterative feedback. The model also struggled with prolonged conversations where essential task requirements were sometimes forgotten, leading to incomplete or inconsistent outputs. Particularly in older languages with limited modern documentation, like Erlang, the model demonstrated weaker performance.

In conclusion, this thesis comprehensively explored ChatGPT-3.5's ability to function autonomously and collaboratively in software engineering tasks. The findings confirm that while ChatGPT-3.5 can effectively assist with diverse programming challenges, its optimal use case lies in collaboration rather than autonomy. Its strengths include accelerating coding workflows, rapid prototyping, and supporting well-specified tasks, while its limitations emerge in complex, poorly documented, or highly abstract problems. Python emerged as the most effective language due to direct interpreter support, while Erlang posed greater challenges due to complexity and limited tool integration.

These insights emphasize ChatGPT's value in enhancing productivity and supporting rapid prototyping while underscoring the need for human oversight in complex, ambiguous, or critical development tasks. The research also highlights the importance of prompt refinement and iterative feedback when working with AI in software engineering. Future research could further explore the ethical and technical implications of AI-assisted coding, such as bias mitigation, improved context retention, and adapting models like ChatGPT-4.0 for deeper debugging and code refactoring tasks. As AI tools evolve, their potential to redefine software engineering workflows will likely expand, offering greater collaborative support for professional developers while emphasizing human-AI synergy in solving complex engineering challenges.

Bibliography

- [1] A. A. A. Adna Beganovic, Muna Abu Jaber. Methods and applications of chatgpt in software development: A literature review. *Southeast Europe Journal of Soft Computing*, 12(1):08–12, March 2023.
- [2] A. Ahmad, M. Waseem, P. Liang, M. Fahmideh, M. S. Aktar, and T. Mikkonen. Towards human-bot collaborative software architecting with chatgpt. In *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, EASE '23, page 279–285, New York, NY, USA, 2023. Association for Computing Machinery.
- [3] M. A. Akbar, A. A. Khan, and P. Liang. Ethical aspects of chatgpt in software engineering research, 2023.
- [4] L. Avila-Chauvet, D. Mejía, and C. O. Acosta Quiroz. Chatgpt as a support tool for online behavioral task programming. *Available at SSRN 4329020*, 2023.
- [5] A. S. Bale, Y. R. Vada, B. E. Oshiojum, U. K. Lakkineni, C. Rao, K. Venkatesh6, and I. Rani. Chatgpt in software development: Methods and cross-domain applications. *International Journal of Intelligent Systems and Applications in Engineering*, 11(9s):636–643, Jul. 2023.
- [6] P. Bera, Y. Wautelet, and G. Poels. On the use of chatgpt to support agile software development. In *Short Paper Proceedings of the Second International Workshop on Agile Methods for Information Systems Engineering (Agil-ISE 2023) co-located with the 35th International Conference on Advanced Information Systems Engineering (CAiSE 2023), Zaragoza, Spain, June 13, 2023*, volume 3414, pages 1–9. CEUR-WS. org, 2023.
- [7] T. B. Brown. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.
- [8] A. Buscemi. A comparative study of code generation using chatgpt 3.5 across 10 programming languages, 2023.
- [9] J. Cao, M. Li, M. Wen, and S. chi Cheung. A study on prompt design, advantages and limitations of chatgpt for deep learning program repair, 2023.
- [10] S. Ekin. Prompt engineering for chatgpt: A quick guide to techniques, tips, and best practices. *TechRxiv*, 2023.
- [11] W. et al. Ethical chatgpt: Concerns, challenges, and commandments, 2023.

- [12] M. Fraiwan and N. Khasawneh. A review of chatgpt applications in education, marketing, software engineering, and healthcare: Benefits, drawbacks, and research directions, 2023.
- [13] M. Fu, C. Tantithamthavorn, V. Nguyen, and T. Le. Chatgpt for vulnerability detection, classification, and repair: How far are we?, 2023.
- [14] M. Gao, J. Ruan, R. Sun, X. Yin, S. Yang, and X. Wan. Human-like summarization evaluation with chatgpt, 2023.
- [15] J. Gibbons and B. C. d. S. Oliveira. The essence of the iterator pattern. *Journal of Functional Programming*, 19:377–402, 2009.
- [16] L. Giray. Prompt engineering with chatgpt: A guide for academic writers. *Annals of Biomedical Engineering*, 51:2629–2633, 2023.
- [17] Q. Guo, J. Cao, X. Xie, S. Liu, X. Li, B. Chen, and X. Peng. Exploring the potential of chatgpt in automated code refinement: An empirical study, 2023.
- [18] M. A. Haque and S. Li. The potential use of chatgpt for debugging and bug fixing. *EAI Endorsed Transactions on AI and Robotics*, 2, May 2023.
- [19] W. Hariri. Unlocking the potential of chatgpt: A comprehensive exploration of its applications, advantages, limitations, and future directions in natural language processing, 2023.
- [20] A. Hörnemalm. Chatgpt as a software development tool: The future of development, 2023.
- [21] M. F. A. Khan, M. Ramsdell, E. Falor, and H. Karimi. Assessing the promise and pitfalls of chatgpt for automated code generation, 2023.
- [22] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474, 2020.
- [23] J. Li, X. Cheng, W. X. Zhao, J.-Y. Nie, and J.-R. Wen. Halueval: A large-scale hallucination evaluation benchmark for large language models, 2023.
- [24] Y. Li, J. Xu, Y. Zhu, H. Liu, and P. Liu. The impact of chatgpt on software engineering education: A quick peek. In *2023 10th International Conference on Dependable Systems and Their Applications (DSA)*, pages 595–596, 2023.
- [25] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Computing Surveys*, 55(9):1–35, 2023.
- [26] Y. Liu, G. Deng, Z. Xu, Y. Li, Y. Zheng, Y. Zhang, L. Zhao, T. Zhang, K. Wang, and Y. Liu. Jailbreaking chatgpt via prompt engineering: An empirical study, 2023.

- [27] Y. Liu, T. Le-Cong, R. Widyasari, C. Tantithamthavorn, L. Li, X.-B. D. Le, and D. Lo. Refining chatgpt-generated code: Characterizing and mitigating code quality issues. *ACM Trans. Softw. Eng. Methodol.*, jan 2024.
- [28] L. LLC. Leetcode. <http://leetcode.com>.
- [29] B. C. S. Loh, A. Y. Y. Fong, T. K. Ong, and P. H. H. Then. Deep learning in digital health with chatgpt: a study on efficient code generation. *European Heart Journal*, 44, 11 2023.
- [30] Y. Lou et al. No more manual tests? evaluating and improving chatgpt for unit test generation, 2023.
- [31] J. Luo, T. Li, D. Wu, M. Jenkin, S. Liu, and G. Dudek. Hallucination detection and hallucination mitigation: An investigation, 2024.
- [32] W. Ma, S. Liu, Z. Lin, W. Wang, Q. Hu, Y. Liu, C. Zhang, L. Nie, L. Li, and Y. Liu. Lms: Understanding code syntax and semantics for code analysis, 2024.
- [33] W. Ma, S. Liu, W. Wang, Q. Hu, Y. Liu, C. Zhang, L. Nie, and Y. Liu. The scope of chatgpt in software engineering: A thorough investigation, 2023.
- [34] N. Marques, R. R. Silva, and J. Bernardino. Using chatgpt in software requirements engineering: A comprehensive review. *Future Internet*, 16(6), 2024.
- [35] N. Nascimento, P. Alencar, and D. Cowan. Comparing software developers with chatgpt: An empirical investigation, 2023.
- [36] M. Neumann, M. Rauschenberger, and E.-M. Schön. We need to talk about chatgpt: The future of ai and higher education. In *2023 IEEE/ACM 5th International Workshop on Software Engineering Education for the Next Generation (SEENG)*, pages 29–32, 2023.
- [37] A. Nguyen-Duc, B. Cabrero-Daniel, A. Przybylek, C. Arora, D. Khanna, T. Herda, U. Rafiq, J. Melegati, E. Guerra, K.-K. Kemell, M. Saari, Z. Zhang, H. Le, T. Quan, and P. Abrahamsson. Generative artificial intelligence for software engineering – a research agenda, 2023.
- [38] S. Omari, K. Basnet, and M. Wardat. Investigating large language models capabilities for automatic code repair in python. *Cluster Computing*, pages 1–15, 05 2024.
- [39] S. Ouyang, J. M. Zhang, M. Harman, and M. Wang. An empirical study of the non-determinism of chatgpt in code generation. *ACM Transactions on Software Engineering and Methodology*, Sept. 2024.
- [40] G. Pu et al. A process-algebraic approach to workflow specification and refinement. *International Conference on Software Composition*, pages 51–65, 2007.
- [41] C. Qin, A. Zhang, Z. Zhang, J. Chen, M. Yasunaga, and D. Yang. Is chatgpt a general-purpose natural language processing task solver?, 2023.

- [42] B. Qureshi. Exploring the use of chatgpt as a tool for learning and assessment in undergraduate computer science curriculum: Opportunities and challenges, 2023.
- [43] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer, 2023.
- [44] L. Reynolds and K. McDonell. Prompt programming for large language models: Beyond the few-shot paradigm, 2021.
- [45] D. Sobania, M. Briesch, C. Hanna, and J. Petke. An analysis of the automatic bug fixing performance of chatgpt, 2023.
- [46] G. Sridhara, R. H. G., and S. Mazumdar. Chatgpt: A study on its utility for ubiquitous software engineering tasks, 2023.
- [47] W. Sun, C. Fang, Y. You, Y. Miao, Y. Liu, Y. Li, G. Deng, S. Huang, Y. Chen, Q. Zhang, H. Qian, Y. Liu, and Z. Chen. Automatic code summarization via chatgpt: How far are we?, 2023.
- [48] N. M. S. Surameery and M. Y. Shakor. Use chat gpt to solve programming bugs. *International Journal of Information Technology: Computer Engineering (IJITC) ISSN : 2455-5290*, 3(01):17–22, Jan. 2023.
- [49] H. Tian, W. Lu, T. O. Li, X. Tang, S.-C. Cheung, J. Klein, and T. F. Bissyandé. Is chatgpt the ultimate programming assistant – how far is it?, 2023.
- [50] C. Waghmare. *Security and Ethical Considerations When Using ChatGPT*, pages 111–132. Apress, Berkeley, CA, 2023.
- [51] S. Wang, H. Scells, B. Koopman, and G. Zuccon. Can chatgpt write a good boolean query for systematic review literature search?, 2023.
- [52] J. White, Q. Fu, S. Hays, M. Sandborn, C. Olea, H. Gilbert, A. Elnashar, J. Spencer-Smith, and D. C. Schmidt. A prompt pattern catalog to enhance prompt engineering with chatgpt, 2023.
- [53] J. White, S. Hays, Q. Fu, J. Spencer-Smith, and D. C. Schmidt. Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design, 2023.
- [54] C. S. Xia and L. Zhang. Keep the conversation going: Fixing 162 out of 337 bugs for \$0.42 each using chatgpt, 2023.
- [55] Q. Zhang, T. Zhang, J. Zhai, C. Fang, B. Yu, W. Sun, and Z. Chen. A critical review of large language model on software engineering: An example from chatgpt and automated program repair, 2024.