UNIVERSIDADE FEDERAL DE MINAS GERAIS
Instituto de Ciências Exatas
Programa de Pós-Graduação em Ciência da Computação

Mauricio Monteiro da Silva

NoCodeGPT: A No-Code Interface for Building Web Apps with Language Models

Belo Horizonte
2025

Mauricio Monteiro da Silva

NoCodeGPT: A No-Code Interface for Building Web Apps with Language Models

Final Version

Thesis presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Prof. Dr. Marco Túlio de Oliveira Valente

Belo Horizonte
2025

Silva, Mauricio Monteiro da.

S586n NoCodeGPT: [recurso eletrônico] / a no-code interface for building web apps with language models. Mauricio Monteiro da Silva. Belo Horizonte – 2025.
1 recurso online (81 f. il., color.) : pdf.

Orientador: Marco Túlio de Oliveira Valente.

Dissertação (Mestrado) - Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação.
Referências: f. 77-81.

1. Computação – Teses. 2. Engenharia de software – desenvolvimento – Teses. 3. Linguagens de programação (Computadores) – Teses. 4. Inteligência artificial – Teses. I. Valente, Marco Túlio de Oliveira. II. Universidade Federal de Minas Gerais Instituto de Ciências Exatas, Departamento de Ciência da Computação. III. Título.

CDU 519.6*32(043)

Ficha catalográfica elaborada pela bibliotecária Irénquer Vismeg Lucas Cruz
CRB 6/819 - Universidade Federal de Minas Gerais - ICEx

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

NoCodeGPT: A No-Code Interface for Building Web Apps with Language Models

MAURICIO MONTEIRO DA SILVA

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores(a):

Prof. Marco Túlio de Oliveira Valente - Orientador
Departamento de Ciência da Computação - UFMG

Prof. Eduardo Magno Lages Figueiredo
Departamento de Ciência da Computação - UFMG

Profa. João Eduardo Montandon de Araújo Filho
Departamento de Ciência da Computação - UFMG

Prof. Henrique Santos Camargos Rocha
Department of Computer Science - Loyola University Maryland

Documento assinado digitalmente
gov.br
HENRIQUE SANTOS CAMARGOS ROCHA
Data: 14/10/2025 12:04:38-0300
Verifique em https://validar.iti.gov.br

Belo Horizonte, 21 de março de 2025.

Documento assinado eletronicamente por **Marco Tulio de Oliveira Valente**, **Professor do Magistério Superior**, em 21/08/2025, às 13:07, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **João Eduardo Montandon de Araujo Filho**, **Professor do Magistério Superior**, em 22/08/2025, às 09:51, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Eduardo Magno Lages Figueiredo**, **Professor do Magistério Superior**, em 15/10/2025, às 17:23, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

A autenticidade deste documento pode ser conferida no site https://sei.ufmg.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **4482107** e o código CRC **B8974F78**.

Referência: Processo nº 23072.251225/2025-41 SEI nº 4482107

Agradecimentos

Meus agradecimentos vão para:

Primeiramente a Deus, por permitir a conclusão deste curso.

A minha familia, pelo apoio e também por estarem sempre ao meu lado nessa caminhada durante este curso de mestrado. Um agradecimento especial a minha esposa que foi minha inspiração e a motivação para fazer este mestrado. Também um agradecimento a minha mãe e ao meu pai que através de suas orações intercederam por mim para que Deus me ajudasse nessa caminhada.

Aos meus amigos, pela amizade, suporte e por todos os momentos incríveis que passamos juntos. Em especial ao Prof. Virgil Del Duca Almeida por ser um bom ouvinte nos momentos difíceis.

Aos meus colegas do Departamento de Controle e Automação do Instituto Federal de Minas Gerais do Campus Betim, pelo companheirismo e contribuição no meu crescimento profissional e pessoal.

Ao meu orientador, Prof. Dr. Marco Túlio de Oliveira Valente, pela paciência, confiança e todos os ensinamentos passados durante esse período.

Aos meus professores, pelo incentivo e conhecimento compartilhado durante o mestrado.

Aos membros da banca, o Prof. Dr. Eduardo Magno Lages Figueiredo, o Prof. Dr. João Eduardo Montandon de Araujo Filho e ao Prof. Dr. Henrique Rocha, pela disponibilidade em participar deste trabalho.

"*O temor do Senhor é o princípio do conhecimento, mas os insensatos desprezam a sabedoria e a disciplina.*"

(Provérbios 1:7)

Resumo

Esta dissertação explora a aplicação de Modelos de Linguagem (LLMs) no desenvolvimento de software, com especial ênfase na construção de software de fim a fim usando o modelo GPT4 da OpenAI. Um experimento exploratório foi realizado para avaliar os benefícios da utilização da interface do usuário ChatGPT durante o desenvolvimento de uma aplicação de software. O estudo envolveu três participantes com níveis variados de proficiência, encarregados de desenvolver um aplicativo web baseado exclusivamente em um aplicativo de referência. O experimento revelou limitações significativas da interface do usuário ChatGPT, particularmente para desenvolvedores com experiência limitada em programação. Ou seja, os resultados indicam que o nível de proficiência do desenvolvedor influenciou significativamente sua capacidade de construir o software. O desenvolvedor novato não conseguiu completar a tarefa com sucesso, enquanto o desenvolvedor mais experiente produziu código de maior qualidade em comparação com os outros participantes. Para enfrentar os desafios encontrados pelos desenvolvedores novatos, uma ferramenta chamada NoCodeGPT foi desenvolvida. Esta ferramenta encapsula o conhecimento sobre tecnologia, ambientes de desenvolvimento e organização de arquivos, isto é, aspectos nos quais os desenvolvedores novatos encontraram maiores dificuldades no nosso estudo exploratório. Assim, o NoCodeGPT visa simplificar o processo de construção de software, permitindo que desenvolvedores novatos criem software fim a fim sem intervenção humana na geração do código da aplicação web. Para avaliar a eficácia do NoCodeGPT, foram realizados dois experimentos com 14 desenvolvedores iniciantes, demonstrando uma taxa de sucesso superior a 60% na construção de dois pequenos sistemas web. Isso indica que NoCodeGPT ajudou a reduzir as limitações enfrentadas pelos desenvolvedores menos experientes, aumentando sua capacidade de criar um aplicativo web mesmo sem conhecimento prévio de programação. O estudo destaca o potencial das ferramentas que utilizam LLMs para democratizar o desenvolvimento de software e melhorar os resultados para desenvolvedores com diferentes níveis de experiência.

Palavras-chave: ciência da computação, engenharia de software, construção de sistemas, modelo de linguagem de larga escala, ChatGPT.

Abstract

This dissertation explores the application of Language Models (LLMs) in software development, with special emphasis on constructing end-to-end software supported by Language Models such as GPT4. An exploratory experiment was conducted to evaluate the benefits of using the ChatGPT user interface during software application development. The study involved three participants with varying levels of proficiency, who were responsible for developing a web application based exclusively on a reference application. The experiment revealed significant limitations of the ChatGPT user interface, particularly for developers with limited programming experience. The results indicate that the developer's level of proficiency significantly influenced his ability to build software. The novice developer failed to complete the task successfully, while the more experienced developer produced higher-quality code than the other participants. To address the challenges faced by novice developers, a tool called NoCodeGPT was proposed. This tool encapsulates knowledge about technology, development environments, and file organization where novice developers encountered the most difficulties during the construction process. The tool aims to simplify the software-building process, allowing novice developers to create end-to-end software without human intervention in generating web application code. Two experiments were conducted with 14 beginning developers to evaluate the effectiveness of NoCodeGPT. The results demonstrated a success rate of more than 60% in end-to-end software construction. This indicates that NoCodeGPT has helped reduce the limitations faced by less experienced developers, increasing their ability to create a web application even without prior programming knowledge. The study highlights the potential of tools that use LLMs to democratize software development and improve results for developers with different levels of experience.

Keywords: science computer, software engineering, construction software, large language model, ChatGPT.

List of Figures

3.1 Screenshot of the main screen (reference implementation). 42
3.2 Diagram (back-end, reference implementation). 42
3.3 Screenshot of the answers screen (Participant P1). 46
3.4 Screenshot of the question list page (Participant P2). 47
3.5 Architecture of the oracle version (with a clear separation between web end-points, business rules, and database persistence). 51
3.6 Architecture of the P1 version, which follows a functional decomposition, i.e, we have modules responsible for each feature, which implement both non-functional (e.g., web end-points) and functional concerns. 51
3.7 Architecture of the P2 version. This is a monolithic architecture where everything is implemented in a single file. 51
3.8 User information page (Oracle). 53
3.9 Answers page, with information about the user in the top (Participant P1). . . 53
3.10 User information page (Participant P2). 53
3.11 Histogram with Cyclomatic Complexity values (back-end). 54
3.12 Histogram with Cyclomatic Complexity measures (front-end). 56

4.1 Page to enter the name and main context of the system under development. . 60
4.2 Prompts for refining generated pages. 61
4.3 NoCodeGPT's high-level architecture. 63
4.4 Front-end and Back-end architecture. 64

5.1 Screenshot of participant P2's application. 67
5.2 Types of prompts implemented by each participant in the pilot experiment (TodoApp). 68
5.3 Rollbacks performed by Participant P2 in the pilot experiment (TodoApp). . . 69
5.4 Question registration page created by participant P7 in the second experiment. 69
5.5 Answer registration page created by participant P7 in the second experiment. 70
5.6 Total prompts used by each participant stacked by type. 71

List of Tables

3.1 Prompts used by each participant. 44
3.2 Number of files. 50
3.3 Number of functions. 52
3.4 Number of lines of code. 52
3.5 Issues by ESLint. 57
3.6 Bugs by SonarQube. 58
3.7 Code Smells by SonarQube. 58

5.1 Participants' experience on web development. 66
5.2 Participants who implemented each user story in the pilot experiment (TodoApp). 67
5.3 Rollbacks by participants in the pilot experiment (TodoApp). 68
5.4 Participants who implemented each user story in the second experiment. . . . 70
5.5 Rollbacks by participants in the second experiment. 71

List of Algorithms

3.1 Authentication function in the version of P1 55
3.2 Example of @typescript-eslint/no-explicit-any in P2 57

Contents

1 Introduction **14**

1.1 Motivation . . . 14
1.2 Overview of the Dissertation . . . 16
1.2.1 Exploratory Study . . . 16
1.2.2 NoCodeGPT Tool . . . 17
1.2.3 Evaluation of the NoCodeGPT Tool . . . 18
1.3 Outline of the Dissertation . . . 18

2 Related Work **20**

2.1 Overview . . . 20
2.2 Large Language Models . . . 21
2.3 Using LLMs to Generate Apps . . . 22
2.4 Using LLMs to Solve Code Challenges . . . 24
2.5 Using LLMs to Support Software Maintainance and Testing . . . 25
2.5.1 Software Maintainance . . . 25
2.5.2 Software Tests . . . 28
2.6 Comprehensive Studies . . . 31
2.7 Security Implications of Using LLMs . . . 34
2.8 Other Applications of LLMs in Software Engineering . . . 36
2.9 Final Comments . . . 39

3 Exploratory Study **40**

3.1 Introduction . . . 40
3.2 Methodology . . . 40
3.2.1 Reference Implementation . . . 41
3.2.2 Participants . . . 42
3.2.3 Inception Meeting . . . 43
3.2.4 Tools Used by the Participants . . . 43
3.2.5 Review Meeting . . . 43
3.3 Experiment Results . . . 44
3.4 Lesson Learned . . . 48
3.5 Threats to Validity . . . 49
3.6 Analysis Using Source Code Metrics . . . 49

3.6.1 Number of Files . . . 49
3.6.2 Number of Functions . . . 52
3.6.3 Lines of Code . . . 52
3.6.4 Cyclomatic Complexity . . . 54
3.7 Static Analysis Tools Evaluation . . . 55
3.7.1 ESLint Results . . . 56
3.7.2 SonarQube Results . . . 57
3.8 Final Remarks . . . 58

4 NoCodeGPT Tool **59**
4.1 Introduction . . . 59
4.2 Features and Architecture . . . 59
4.3 Main Features . . . 60
4.4 Architecture and Implementation . . . 62
4.5 Final Remarks . . . 64

5 NoCodeGPT Evaluation **65**
5.1 Introduction . . . 65
5.2 Methodology . . . 65
5.3 Pilot Experiment Results (TodoApp) . . . 67
5.4 Results of the Second Experiment (ForumApp) . . . 68
5.5 Participants' Perceptions . . . 72
5.6 Lessons Learned . . . 73
5.7 Final Remarks . . . 74

6 Conclusion **75**
6.1 Contributions of the Dissertation . . . 75
6.2 Future Work . . . 76

References **77**

Chapter 1

Introduction

1.1 Motivation

In recent years, Large Language Models (LLMs) have emerged as one of the most significant innovations in the field of Artificial Intelligence, with a profound and growing impact on various aspects of our modern society [36]. These models, which include architectures such as GPT-3 [5] and BERT [8], are capable of understanding and generating text with fluency and coherence that rivals human capacity. Its applicability extends from virtual assistants and chatbots to the generation of creative content and machine translation, transforming the way we interact with technology and among ourselves.

In Software Engineering (SE), LLMs have demonstrated a considerable potential to impact established practices and processes. The ability of these models to process and generate natural language makes them valuable tools for automatic documentation, requirements analysis, and even code generation [41]. In addition, they have been exploited to improve the efficiency of tasks such as software testing, where they can generate test cases or identify failures in systems [42].

Despite its potential, applying LLMs in Software Engineering has predominantly focused on specific and smaller-scale tasks. Most current implementations use these models to solve particular problems, such as bug fixes or code refactoring, rather than addressing the development of complete systems. This is partly due to the complexity inherent in large-scale software development, which involves writing code and integrating multiple components, managing requirements, and ensuring quality.

The quality and relevance of LLMs output are also closely tied to the specificity and clarity of the prompts provided by users [7]. This dependency presents a notable challenge for novice developers who may lack the skills to create detailed and informative prompts [20], thereby limiting the effectiveness of AI-assisted development.

The use of Large-scale Language Models (LLMs) is increasingly being adopted by software developers, with some studies indicating important improvements in productivity. For instance, a recent study by GitHub, based on telemetry data from nearly one

million users, found that developers tend to accept 30% of the code suggestions provided by the Copilot tool.[1] The study extrapolates that this adoption rate is equivalent to adding 15 million developers to the global workforce of software professionals. It also concludes that these productivity gains will have a significant impact as developers leverage AI for solution design and accelerate digital transformation worldwide.

Other studies have explored the usage and benefits of language models in specific software engineering tasks, such as fixing bugs [40, 45], writing unit tests [39], writing code comments [11, 38], and solving programming problems [26, 30]. However, to the best of our knowledge, there are few papers that investigate the use of language models for the end-to-end construction of software systems. This involves contexts where a developer has a set of requirements and must design, implement, test, and validate an entirely new system. In such scenarios, language models are used as code generators, receiving prompts that describe the functional and non-functional requirements of a system and producing a runnable software application as output. One exception is a paper by Peng *et al.* [34], where the authors use GitHub Copilot to implement a web server starting from a high-level textual description. However, this implementation is relatively small (as it is a simple web server) and is performed entirely in JavaScript. It does not follow widely adopted software architectures, such as those organized into front-end and back-end components, which are prevalent in modern web-based systems. The implementation also does not utilize popular frameworks for building web interfaces or database systems. Another example is a demonstration conducted by one of the co-founders of OpenAI during the launch of GPT-4.[2] To showcase the capabilities of the new version, he drew a mockup of an application on a napkin, and from a photo of that mockup, ChatGPT was able to generate a functional web application, although it appeared to consist only of its front-end component.

The existing literature, although rich in examples of LLMs applications for specific SE tasks, reveals a significant gap: the lack of tools and methodologies that harness the power of LLMs to support the end-to-end development of Web systems. While studies such as Chen *et al.* [7] explore the use of LLMs for code generation, and others, like Ahmad et al. [2], investigate its application in software testing, there is a lack of research that integrates these capabilities into a cohesive workflow for the development of complete systems.

This gap represents a promising opportunity for research. Developing tools that enable the use of LLMs in the construction of web systems would extend the scope of these models, but could also change the way systems are designed and implemented. By exploring this direction, we could not only increase the efficiency and quality of software

[1] https://github.blog/news-insights/research/the-economic-impact-of-the-ai-powered-developer-lifecycle-and-lessons-from-github-copilot/

[2] https://www.youtube.com/live/outcGtbnMuQ

development but also facilitate access to advanced technologies, enabling developers of all skill levels to create robust and scalable systems.

In summary, the motivation for this research lies in the need to contribute to research and developing solutions that integrate LLMs into the development process of Web systems in a comprehensive and effective way. In doing so, we hope not only to contribute to the advancement of the state of the art in Software Engineering but also to open new possibilities for the application of artificial intelligence in practical and large-scale contexts.

1.2 Overview of the Dissertation

The main goal of this dissertation is to investigates solutions in the field of software engineering for end-to-end implementation of Web systems, using ChatGPT as a support tool. Particularly, our focus is to explore ChatGPT's ability to build an end-to-end web system, evaluating its efficiency in generating code for front-end, back-end, and database architecture modules. The research was structured in three main stages, each addressing a key aspect of the development and evaluation of Web systems with the aid of language models.

1.2.1 Exploratory Study

The first study carried out in this dissertation had as main objective to evaluate the use of ChatGPT in web systems development, focusing on its capabilities and limitations in programming tasks and system design. The methodology adopted involved the use of ChatGPT in realistic development scenarios, allowing a qualitative analysis of the results obtained. This study was essential to identify both the potential and challenges associated with ChatGPT in the context of software development.

Initially, we developed a small web-based system using specific technologies and six user stories. This system served as a benchmark to establish the minimum requirements expected for building a similar application. From this point of reference, we asked three participants in an exploratory study to evaluate the benefits and limitations of using ChatGPT in the process of building a web application.

During the study, each participant used the user interface provided by ChatGPT

to build an application that met the same requirements of the reference implementation. Analysis of the construction process of each participant revealed that novice developers faced significant difficulties, which impacted the development process. This finding highlights the need for human supervision and the importance of a clear understanding of ChatGPT's limitations, especially for developers with less experience.

In addition, the study included an evaluation of the code generated by ChatGPT, to determine the quality level of the produced code. The results indicated that, although ChatGPT can accelerate development and offer creative solutions, the accuracy of the generated code does not always meet the expected standards, reinforcing the need for revision and adjustments by experienced developers.

Thus exploratory study provided valuable insights into the use of ChatGPT in web systems development. It highlighted both the potential of the tool to accelerate processes and foster creativity as well as the challenges related to accuracy and the need for human supervision. These learnings are fundamental to guide future research and practices in the use of language models in software engineering.

1.2.2 NoCodeGPT Tool

The design and implementation of NoCodeGPT represents the second essential step of this research, focusing on simplifying the development of web systems and eliminating the barriers faced by novice developers. NoCodeGPT was designed and implemented to eliminate the need for advanced programming knowledge, allowing users without prior experience to develop complete web applications efficiently and intuitively.

The motivation for the design of NoCodeGPT tool came from our exploratory study, which identified significant difficulties faced by novice developers, negatively impacting the development process. To mitigate these difficulties, the NoCodeGPT tool was designed with a set of features that simplify the process of building web systems. Among its main functionalities, the ability to automatically generate code for the front-end and back-end directly in the project stands out, significantly reducing the need for manual intervention. In addition, the tool offers predefined templates to streamline construction and reduce the time required to implement web systems.. This functionality is crucial to facilitate development, focusing on building applications from prompts that describe the requirements desired by the user.

Moreover, NoCodeGPT's user interface is designed to be intuitive and friendly, making it easy for the user to interact with. The web allows users to request changes through prompts and then view requested changes, being able to customize their ap-

plications according to their needs and expected requirements, without the traditional complexity associated with software development.

1.2.3 Evaluation of the NoCodeGPT Tool

The evaluation of the NoCodeGPT tool was conducted as the final step of our research, to determine its effectiveness and usability in the context of web systems development. For this, we performed two distinct experiments involving participants who had a proficiency profile of beginners in the exploratory study. Both experiments were designed to evaluate whether NoCodeGPT could overcome the main barriers identified during the exploratory study.

The results of the experiments were promising. NoCodeGPT has proven to be an effective tool for facilitating web system development, allowing novice developers to build a web system. Participants reported that the tool was intuitive and easy to use, even for those with limited or no programming experience. In addition, the quality of the generated code was considered satisfactory, with only small improvements suggested by participants. The overall satisfaction of users was high, with many highlighting the potential of the tool in reducing the dependence on advanced knowledge and also being able to speed up development time.

1.3 Outline of the Dissertation

The dissertation is organized in a five-chapter structure including this chapter. The chapters that make up this dissertation are:

The Chapter 2 explores the related work and presents the fundamental concepts about Large-scale Language Models (LLMs) and their application in software engineering seeking to focus on software construction. These concepts form the theoretical basis on which the dissertation develops.

In the Chapter 3, we detail our exploratory study, which evaluates the benefits and limitations of using ChatGPT's user interface in the software building process. This chapter provides a critical analysis of the current capabilities and gaps that exist when using ChatGPT.

Chapter 4 presents the NoCodeGPT tool, developed to overcome the barriers iden-

tified in our exploratory study, which impacted novice developers by using only the ChatGPT interface for software development.

Chapter 5 describes two experiments conducted with 14 novice developers, assessing whether NoCodeGPT effectively improves development capabilities and mitigates the obstacles faced by these developers.

Finally, Chapter 6 closes the master's thesis, offering final considerations and suggesting directions for future research.

Chapter 2

Related Work

2.1 Overview

The ChatGPT model, developed by OpenAI, is a language model that employs artificial intelligence to generate text-based responses interactively, simulating a conversation with a user. It is based on the Generative Pre-trained Transformer (GPT) architecture, which leverages a vast amount of textual data available on the internet for its training.

The training process involves a machine learning technique known as "unsupervised learning", as described by Müller *et al.* [27]. In this process, the model is exposed to a large corpus of text, allowing it to learn linguistic patterns and general knowledge across a wide range of topics. This enables the model to provide information and perform more advanced tasks. According to Ouyang *et al.* [32], the model undergoes "fine-tuning" for specific tasks using a large set of dialogue examples, where an input (user's question or message) and an output (model's response) are provided. Through these examples, the model learns to map an input to an appropriate response.

Brown *et al.* [5] describes that during text generation, ChatGPT operates based on probabilities, attempting to predict the next word or phrase using the given context. It utilizes the context provided by the user to generate relevant responses, predicting the next word or phrase based on prior context and available information. He affirms that the model is also capable of leveraging general information and prior knowledge due to its training on a diverse array of texts, allowing the ChatGPT to produce coherent and relevant text in response to an input prompt, using previous contexts to understand and generate contextually appropriate responses.

Radford *et al.* [35] highlights the importance of knowing that ChatGPT not only replicates information but also creates original text, making it useful for a variety of applications such as text generation, question answering, and more. However, ChatGPT may occasionally produce incorrect, confusing, or unsatisfactory responses. Therefore, it is advisable to critically evaluate the generated responses and consider consulting additional sources for accurate and reliable information.

This chapter provides a comprehensive overview of works related to our master's dissertation. In Section 2.2, we explore research papers on Large Language Models (LLMs). Section 2.3delves into studies concerning the use of LLMs for app generation. In Section 2.4, we examine research focused on employing LLMs to solve code challenges. Section 2.5 reviews studies on using LLMs to support software maintenance and testing. This section is further divided into two subsections: Subsection 2.5.1 concentrates on papers utilizing LLMs in software maintenance, while Subsection 2.5.2 highlights research employing LLMs in software testing. Section 2.6 reviews comprehensive studies, followed by Section 2.7, which discusses the security implications of using LLMs. Section 2.8 covers papers on other applications of LLMs in software engineering. The chapter concludes with Section 2.9, which provides final comments.

2.2 Large Language Models

The GPT-4 Technical Report by OpenAI [1] highlights significant advancements in large language model technology, particularly in the realm of code generation. As an evolution of its predecessor, GPT-3, GPT-4 showcases enhanced capabilities in natural language processing tasks, which extend to generating code. The report details improvements in GPT-4's architecture, featuring a larger and more optimized parameter set that allows it to excel in various benchmarks, including those related to code generation. The Report by OpenAI highlights GPT-4's ability to handle complex prompts and logical reasoning is particularly beneficial for code generation tasks, where understanding intricate instructions and producing accurate, functional code is crucial. The model's improvements in factual accuracy and consistency address previous challenges such as hallucination and inconsistent output, making it more reliable for technical applications, including software development and programming assistance. Furthermore, OpenAI emphasizes safety and alignment strategies in GPT-4, which are critical for code generation. By reducing bias and preventing harmful outputs, GPT-4 is better aligned with user intentions, ensuring that generated code is not only functional but also ethical and secure. These enhancements make GPT-4 a powerful tool for developers, offering reliable support in generating high-quality code across various programming languages and contexts.

The study conducted by researchers at Meta [28], presents CodeCompose, an AI-assisted code authoring tool developed and implemented internally at Meta. CodeCompose is based on the InCoder language model, which combines generative abilities with bidirectionality, meaning it can suggest entire code blocks or statements during the code creation process. This tool has been scaled to serve tens of thousands of developers at

Meta, spanning over ten programming languages and various coding platforms. The study discusses the unique challenges that arise when deploying such tools in large industrial environments, including user experience and performance metrics. The research highlights a 22% acceptance rate of CodeCompose suggestions across multiple languages, with 8% of total user-written code being derived from these accepted suggestions. Metrics from the large-scale deployment of CodeCompose reveal its impact on internal code authorship at Meta over a 15-day period, during which CodeCompose offered 4.5 million suggestions. Qualitative analysis results show overwhelming positive reception, with 91.5% of participants viewing CodeCompose favorably. The tool not only aids in code creation but also encourages developers to produce more code documentation and assists them in discovering new APIs. Looking to the future, the researchers outline plans to enable additional features, including enabling block completions, where developers can receive suggestions for larger chunks of code and more interaction modalities, such as an IDE-integrated conversational interface.

2.3 Using LLMs to Generate Apps

Recent studies in software engineering have increasingly focused on reducing dependency on traditional coding through low-code and no-code platforms, aiming to bridge the gap between technical and non-technical users. Research by Hagel, Hili, and Schwab [13] notably discusses methods for transforming low-code platforms into true no-code environments, enabling users without formal programming knowledge to build sophisticated applications. Their work identifies that the effective integration of domain-specific language (DSL) prompts with LLMs is a critical step toward achieving this transformation. By tailoring prompts to specific tasks, LLMs can better interpret user intentions, reducing errors and optimizing output for non-programmer-friendly applications. This aligns with the trend toward more accessible development processes, as discussed in foundational studies in the field of visual programming and end-user development.

Harter's [14] research on the use of LLMs for developing Human-Machine Interfaces (HMIs) within no-code frameworks emphasizes the importance of accuracy and reliability in LLM output for environments where errors may lead to significant risks. The integration of LLMs with existing application programming interfaces (APIs) in such domains facilitates the automatic generation of interfaces, reducing the need for developer intervention and ensuring adherence to regulatory standards without extensive programming requirements. The research thus points toward potential frameworks that would allow more widespread adoption of no-code solutions even in sensitive sectors.

A study with specific context conducted by Le and Zhang [22] describes their research on leveraging ChatGPT to automatically implement log file analyzers. The study evaluates ChatGPT's ability to perform automated log analysis, focusing on two research questions: (1) Can ChatGPT effectively analyze logs? (2) How does ChatGPT perform with different prompting methods? This work stands as one of the initial investigations into ChatGPT's analytical capabilities, revealing that ChatGPT is particularly effective when provided with few-shot prompting. This approach involves giving ChatGPT small examples to guide its learning in performing log analysis tasks, which appears to enhance its performance. Three metrics were employed to evaluate the results: Group Accuracy, Message Level Accuracy, and Edit Distance. For Group Accuracy, ChatGPT achieved top precision in three of the 16 datasets analyzed, with an average precision of 0.721. They used baselines in the three metrics to compare the effectiveness of ChatGPT in log analysis tasks, using five state-of-the-art log analysis tools as baselines: AEL [18], Spell [9], Drain [15], Logram [23] and SPINE [43]. This result surpasses Logram's [23] average performance by 33.5% and reaches 85% of the effectiveness of the best benchmark solution, Drain [15]. In terms of Message Level Accuracy and Edit Distance, ChatGPT significantly outperformed other solutions, with improvements ranging from 41.0% to 212.1% in Message Level Accuracy and 14.2% to 50.2% in Edit Distance. ChatGPT excelled in 10 of the 16 datasets for Message Level Accuracy and 8 of the 16 for Edit Distance. This indicates its capability to differentiate variable tokens within log messages, as evidenced by high Message Level Accuracy values. However, ChatGPT faced challenges correctly identifying specific log information, which varies significantly across data (such as domain URLs, API endpoint addresses, block/task IDs, etc.), leading to lower Group Accuracy scores. The document also provides examples of log models generated by both ChatGPT and the Drain, highlighting ChatGPT's strength in accurately identifying variable values and types, though noting its difficulty in recognizing certain specific log information as single variables.

Guo [12] explores the growing interest in AI tools such as ChatGPT and GitHub Copilot within the realm of software development. These tools have been praised for their ability to assist with coding tasks, yet their effectiveness in real-world programming scenarios remains a subject of ongoing debate. Numerous blog posts and articles have documented initial experiences with these AI tools, often emphasizing their potential to address small, self-contained programming problems. However, transitioning from handling isolated tasks to managing comprehensive software projects presents a significant challenge. Guo also highlights the development of Swift Papers, a project designed to integrate AI into a realistic programming task, serving as a case study for examining the capabilities of AI tools. The project involved creating a Chrome extension that plays music from the year a research paper was published, utilizing AI to assist in design and implementation. The creation of the extension underscored the challenges of relying solely

on AI for complex tasks, such as setting up development environments, managing permissions, and integrating third-party APIs. Throughout the project, the limitations of AI tools in providing comprehensive guidance and context-aware assistance became apparent. While ChatGPT offered valuable insights and code suggestions, it often defaulted to outdated information and required explicit prompts to generate optimal solutions. This highlights the necessity of human expertise in navigating the intricacies of real-world programming tasks. Guo asserts that advancements in AI tools should focus on enhancing their ability to teach and guide users through complex software projects.

2.4 Using LLMs to Solve Code Challenges

The study conducted by Dakhel *et al.* [26] aimed to evaluate GitHub Copilot's ability to generate solutions for fundamental algorithmic problems and compare them to those created by human programmers. The research found that GitHub Copilot can provide solutions for most fundamental algorithmic issues, although some solutions may contain errors. The study noted challenges in the integration of different methods, yet the quality of GitHub Copilot's suggestions matched that of humans and required less effort for correction. However, it cautioned that GitHub Copilot might not be as effective for novice developers, who may struggle to distinguish between the tool's correct and incorrect suggestions. The research emphasizes GitHub Copilot's value as a programming resource, especially for participant developers. For future development, the researchers suggest creating a tool to filter out erroneous and suboptimal suggestions from GitHub Copilot to enhance its utility in software development contexts.

2.5 Using LLMs to Support Software Maintainance and Testing

2.5.1 Software Maintainance

The study Koh [21] presents a workflow using LLMs to automate the creation of design structure matrices, crucial in engineering and project management. This approach supports the construction of complex models by non-technical users, highlighting the LLMs' ability to simplify traditionally complex workflows and reduce dependency on specialized skill sets. Koh's study emphasizes that the implementation of LLMs for such tasks not only democratizes access but also paves the way for greater innovation in workflow automation.

The potential of Large Language Models (LLMs) to comprehend and generate code within a defined context is advancing the concept of LLMs as "co-pilots" in development tasks. Studies show that such models can be trained or prompted to produce accurate code suggestions, assist in debugging, and generate functionality descriptions, thereby easing the development process. For instance, Nam, Macvean, and Hellendoorn [29] highlight how LLMs can parse entire codebases or individual code segments to assist developers with limited technical backgrounds, thus furthering the potential for truly no-code solutions. Additionally, their findings on prompt optimization underscore the importance of prompt quality and specificity, a recurring theme in LLM-related research, especially for applications within no-code platforms.

The study by Busch, Bainczyk, and Steffen [6] explores the application of Large-scale Language Models (LLMs) in software generation within a Language-Driven Engineering (LDE) framework, particularly emphasizing NoCode/LowCode development scenarios. Their approach integrates LLM-based code generation with domain-specific languages (DSLs) to break down complex tasks into smaller, more manageable components. This strategy enables LLMs to generate code for less complex tasks, while other tasks are addressed using textual or graphical DSLs. A practical example provided in the study is the migration of a point-and-click adventure generator from JavaScript to TypeScript. This migration is facilitated by decomposing tasks into smaller segments, where game logic is articulated in natural language, and the foundational code is generated from graphical models. The validation of the migrated system is conducted through automata learning and model analysis, allowing for the verification of migration accuracy by comparing the outputs of the original and migrated systems. The approach underscores the capability of

LLMs to automatically incorporate type information necessary for TypeScript and maintain automatic validation through automata learning, thereby ensuring the reliability of the generated code.

The study realized by Liu *et al.* [24] delves into the quality and reliability of code generated by ChatGPT in Java and Python, utilizing a dataset of 2,033 programming tasks sourced from LeetCode, which vary in difficulty. The research identifies several key factors that influence the performance of the generated code. One significant factor is the difficulty level of the task, which directly affects the quality of the output. Additionally, the timing of when a task was introduced plays a crucial role, with a noticeable decline in performance for tasks introduced after January 2022. The size of the program is another important factor, as it impacts the quality of the generated code. The study analyzed 4,066 programs generated by ChatGPT. Where 2,756 programs were classified as correct, whereas 1,082 produced incorrect outputs. Additionally, 177 programs encountered compilation or execution errors, and 1,930 exhibited maintainability issues. The study highlights several common problems in the generated code, including errors, incorrect outputs, style and maintainability concerns, and performance issues. These challenges underscore the difficulties of relying solely on AI-generated code for practical applications. Moreover, the study assessed ChatGPT's ability to self-correct. It was found that the quality of the code could improve by 20% when incorporating feedback from static analysis tools. Despite this improvement, limitations persist, indicating that while ChatGPT can enhance code quality to an extent, further refinement is necessary to ensure its reliability in practical applications.

The study conducted by Nguyen *et al.* [30] involved an empirical analysis of the accuracy and comprehensibility of synthesized code solutions provided by GitHub Copilot, using 33 LeetCode problems as a reference. The Copilot's suggestions were evaluated for languages like Java, JavaScript, Python, and C. The correctness of the proposed solutions was tested on LeetCode, while comprehensibility was assessed using cognitive complexity and cyclomatic complexity metrics. The findings indicated that Python suggestions closely matched GitHub's internal assessment of Copilot. Java suggestions achieved the highest accuracy score at 57%, in contrast to JavaScript solutions, which had the lowest score at 27%. Moreover, the study highlighted that Copilot-generated suggestions generally exhibited low complexity, with no statistically significant differences in complexity scores across different languages for the same solution. Potential deficiencies in Copilot were identified, including the generation of code segments that could be further simplified or compacted and reliance on undefined auxiliary methods. LeetCode was chosen as the testing platform for Copilot's suggestions as it provides the necessary context to formulate Copilot queries and evaluate solution accuracy through test cases. To measure the comprehensibility of the suggestions, SonarQube was employed to calculate cognitive and cyclomatic complexities, both of which positively correlated with code comprehensibility.

In summary, the investigation revealed that Copilot suggestions are characterized by low complexity, with no significant statistical differences in complexity scores for the same solution across different programming languages. Additionally, certain limitations such as the potential for code simplification or compactness and the inclusion of undefined auxiliary methods were identified in Copilot's generated code.

In their study, Mastropaolo *et al.* [25] explored the robustness of GitHub Copilot by investigating whether semantically equivalent descriptions would lead to the same code recommendations from Copilot. The research involved generating 892 Java methods from original descriptions and then creating semantically equivalent descriptions both manually and automatically. The equivalent descriptions were generated manually and using automated tools such as PEGASUS and TP. PEGASUS, a deep learning-based tool for automated paraphrasing, and TP, a heuristic-based approach, were utilized to create equivalent descriptions of the functions. These descriptions were then used to test the robustness of GitHub Copilot's ability to recommend code based on them. The original and equivalent descriptions were input into Copilot, and the researchers analyzed the percentage of cases where equivalent descriptions resulted in different code recommendations, as well as the impact on the recommendation quality. Additionally, they employed approved tests as indicators of the correctness of the recommendations generated by Copilot. The study revealed that altering the description resulted in different code recommendations in approximately 46% of the cases. Furthermore, it was found that variations in the equivalent descriptions could affect the correctness of the generated code by about 28%. The researchers concluded that the input description plays a critical role in code recommendation, emphasizing the need to test and enhance the robustness of deep learning-based code generators.

The study conducted by Geng *et al.* [11] investigates the use of Large Language Models (LLMs) for generating code comments that cater to multiple intentions, leveraging the paradigm of in-context learning. In-context learning is a paradigm where a large language model (LLM) is provided with a prompt that includes a natural language instruction, a handful of examples demonstrating how the task should be performed, and a query that needs to be addressed. The model uses this context to generate a response to the query without any parameter updates or additional training. This approach leverages the knowledge the model has acquired during its pre-training on large-scale datasets, allowing it to generalize well to new tasks with minimal examples. The primary aim of the research is to explore the potential of LLMs, such as Codex, in effectively producing code comments without the need for specific training or fine-tuning for this task. The study evaluates Codex's ability to generate multi-intent comments through zero-shot, one-shot, and few-shot learning. The focus is on determining whether basic in-context learning can compete with or surpass advanced methods. The study details the employed methodology, which includes defining a model for multi-intent comment generation, a

demonstration retrieval approach to better guide the model on expected behavior, and reranking strategies aimed at improving the quality of generated comments. The study concludes by highlighting the positive outcomes of the proposed approaches. Specifically, it demonstrates that when Codex is provided with a few examples (specifically ten or more) as prompts, it can generate code comments that are more accurate and useful than those produced by the latest supervised learning methods. This finding paves the way for future research into using LLMs in multi-intent code comment generation tasks and suggests potential improvements for the methodologies employed.

Research conducted by Shin *et al.* [38] examined the efficiency of the generative LLM GPT-4 compared to 18 other similar models across three standard AI-assisted Software Engineering tasks: code generation, code summarization, and code translation. To assess the effectiveness of GPT-4, the researchers employed three different prompt engineering techniques, which are methods of providing instructions to the model. The study indicates that GPT-4 does not necessarily outperform the other models in all tasks. In code commenting, GPT-4 surpassed the top-ranked fine-tuned model by an average of 8.33 percentage points in BLEU, a metric used to measure the effectiveness of automatic translations. However, in code generation, the top-ranked fine-tuned model outperformed GPT-4 by an average of 16.61 and 28.3 percentage points in BLEU. In code translation, there was a tie between GPT-4 and other fine-tuned models. The study also included feedback from 27 postgraduate students and 10 industry professionals, who assisted in evaluating the quality of GPT-4's responses. It was observed that GPT-4's performance significantly improved when prompts were interactive, meaning human feedback and instructions were exchanged back and forth with the AI. The findings suggest that GPT-4 holds significant potential for software engineering tasks but still requires human interaction to maximize its performance and needs further research and improvements.

2.5.2 Software Tests

The research conducted by Siddiq *et al.* [39] investigates the capabilities of Large Language Models (LLMs) such as GPT-3.5, Codex, and CodeGen in generating unit tests within the context of Test-Driven Development (TDD). The primary goal of this study was to assess the effectiveness of these LLMs in producing unit tests and to examine how different contextual styles influence their performance in this task. To address critical questions regarding the efficiency of LLMs in generating unit tests, the study was structured around two key research questions. The first question focused on evaluating the ability of LLMs to generate tests. In contrast, the second aimed to identify specific

elements of a code context that could impact the success of creating these unit tests. To explore these research questions, the study employed a rigorous methodology, which involved selecting code from Java projects tested with JUnit5 and using three different LLMs to generate unit tests. The analysis included data collected from implementations of solutions from AWS Labs' HumanEval and 47 projects from the EvoSuite SF110 benchmark, selecting specific classes and methods for test generation. The LLMs were tasked with creating 10 test cases for each method, using distinct strategies due to limitations such as token count. The analysis of the data generated by the LLMs revealed multiple facets, including test compilation, coverage, correctness, and the presence of "test smells". A tool called TsDetect was used to identify common issues in tests, known as "test smells". The study's findings highlighted the relatively inferior performance of LLMs compared to EvoSuite across various dimensions. Notably, EvoSuite was able to produce a unit test for each of the 160 classes under test, whereas ChatGPT 3.5 managed only 130 compilable unit tests. Additionally, it was observed that LLMs, in some instances, tended to "hallucinate" types, methods, or paths that did not exist during test generation. The study concludes that while LLMs demonstrated performance close to that of EvoSuite on the HumanEval dataset, their performance was considerably worse on the open-source projects of EvoSuite, particularly concerning coverage. LLMs still have a significant journey ahead before becoming robust and reliable tools for unit test generation in TDD environments. However, the research underscores the potential of these technologies as starting points in the TDD process and suggests directions for future work aimed at improving LLMs to better understand language semantics, thereby increasing test correctness and compilation rates.

The study of Xia *et al.* [45] aims to enhance the efficiency of Automated Program Repair (APR), which is used to automatically fix software issues. Traditional APR methods based on machine learning face the challenge of a limited variety of fixes, rendering them ineffective for addressing complex bugs. This limitation stems from their reliance on bug-fix datasets for modeling corrections or predicting possible solutions. The researchers utilized Large Language Models (LLMs), trained with billions of text/code tokens, instead of depending solely on limited bug-fix datasets. They explored the use of nine advanced LLMs for APR. The study applied three different repair configurations to the LLMs, designed to evaluate their potential in generating software fixes:

1. Generation of a complete fix function;
2. Filling in a code snippet with a prefix and suffix;
3. Generation of a single-line fix.

These configurations were tested across five different datasets in three different programming languages. The study's results were quite positive, demonstrating that

the advanced LLMs significantly outperformed all existing APR techniques on the tested datasets. The researchers concluded that, generally, larger models tend to perform better. Additionally, the LLMs produced patches with higher compilation rates, proving efficient in patch classification and verification of their correctness. The researchers suggest that the performance of LLM-based APR can be further improved by increasing the sample size and incorporating information from correction models. The study indicates a promising future for the use of LLMs in automatic program repair, highlighting significant benefits such as the ability to fix a greater number of bugs, faster patch generation speeds, and higher compilation rates compared to existing APR techniques.

The research conducted by Sobania *et al.* [40] explores the capability of ChatGPT in identifying and repairing errors in programming code. Aimed at assisting software developers in bug detection and correction, various automated program repair (APR) techniques have been introduced, many of which are based on deep learning approaches. The study evaluates ChatGPT's performance using the QuixBugs benchmark set, which consists of challenging yet small problems in terms of lines of code, all written in Python. According to the authors, the primary innovation of ChatGPT is its dialogue system, which allows the inclusion of additional information (such as expected outputs or observed error messages), potentially increasing its success rate in bug correction. The analysis compares ChatGPT's performance with other DL-based APR approaches, such as CoCoNut and Codex, as well as with standard APR methods reported in the literature. The results indicate that ChatGPT's performance is competitive, successfully correcting 19 out of 40 bugs from the benchmarks without additional information and achieving an even higher success rate, correcting 31 out of 40 bugs, when extra information is provided through its dialogue system. This performance surpasses both DL approaches like CoCoNut and Codex and standard APR methods, revealing the potential of large-scale language models for programmatic repair when combined with interactive dialogue. The study demonstrates that while ChatGPT is effective in detecting and correcting simple bugs, it struggles with more complex errors involving intricate programming logic. To better illustrate their findings, the authors categorized the responses generated by ChatGPT and conducted brief dialogues with the model, providing pertinent observations about the work performed. The responses were classified into various categories such as "More Information Needed", "No Bug Found", "Correct Fix Provided", "Attempts to Fix Something Else", "Provides a Solution but Introduces a New Bug", and "Alternative Implementation". Overall, the study concludes that ChatGPT can be an effective tool for automatic program repair, especially when utilizing its dialogue capability to provide more context about the bugs.

TestGen-LLM is an automated testing tool developed by Meta, that leverages two Large Language Models to enhance unit test case generation. A study conducted by Alshahwan *et al.* [3] evaluated the tool's effectiveness in real-world software develop-

ment. The research involved three test-a-thons, revealing several key insights. Notably, TestGen-LLM improved 196 out of 1,979 test classes, which translates to approximately 10% enhancement. Developers accepted 73% of the tool's recommendations, indicating a high level of trust and usability. Furthermore, 75% of the generated tests were constructed correctly, and 57% of these tests passed reliably, showcasing the tool's capability to produce functional test cases. Additionally, there was a 25% increase in line coverage, highlighting the tool's contribution to more comprehensive testing. The tool collaborates with human engineers, who are crucial in reviewing and validating the generated tests to ensure they adhere to coding standards. This collaborative approach underscores the potential of automated testing tools to augment human expertise, rather than replace it while maintaining code quality and preventing regression issues. However, the performance of TestGen-LLM varied between the Facebook and Instagram platforms, primarily due to differences in the available training data. In examining the performance of TestGen-LLM across different platforms, notable variations were observed between Facebook and Instagram, primarily attributed to disparities in the available training data. The success rate for generating new test cases was marginally higher for Facebook, at 0.05, compared to Instagram's 0.04. Interestingly, Instagram had a greater number of clinical trials, totaling 23,535, in contrast to Facebook's 8,996. However, Facebook's training data included approximately ten times more examples of human-written test cases than Instagram's, likely contributing to its superior success rate. thesee findings underscore the significant influence that both the quantity and quality of training data have on the efficacy of automated testing solutions like TestGen-LLM. The results suggest that platforms with more comprehensive and diverse training datasets may achieve better performance in generating accurate and effective test cases.

2.6 Comprehensive Studies

Sadik *et al.* [37] research aimed to evaluate the use of ChatGPT as an intermediary between programmers' natural language and machine code, identifying its benefits and limitations. The study's results indicate that while large language models (LLMs) can save time and improve efficiency in software development, they are not yet equipped to replace human programmers entirely. The areas addressed include code generation, documentation, bug detection and correction, code refactoring, and step-by-step execution of source code, among other applications. The study identifies challenges such as handling complex logic, inadequate short-term memory, vulnerability to adversarial inputs, and an inability to learn new content in real time. Additionally, it warns of the risks as-

sociated with over-dependence on ChatGPT-generated solutions, including the potential generation of biased or useless code, alongside concerns regarding security, environmental impact, and the economic and legal implications of using such technologies. It was also noted that while several areas require improvement, Codex models surpass GPT models in accuracy when solving programming problems.

A collaborative study conducted by researchers from Microsoft Research, GitHub, and MIT [34] examined the impact of Artificial Intelligence (AI) on developer productivity. Participants in the study were software developers tasked with implementing an HTTP server in JavaScript as quickly as possible. These developers were split into two groups: one had access to the AI tool GitHub Copilot, while the other worked without such assistance. The study found that the AI tool had a significant effect on the developers' productivity, enabling them to complete the assigned task 55.8% faster. This suggests that the tool is particularly beneficial for less experienced programmers, aiding in their professional development. However, it is crucial to note that the research did not evaluate the quality of the code produced with the assistance of GitHub Copilot, leaving room for further studies to explore this aspect. Additionally, there is a need to assess the performance of AI tools across various tasks and programming languages. The researchers speculate that if these findings could be generalized to the entire developer population, there would be a substantial increase in productivity, would imply a significant amount of cost savings in the economy, and have a notable impact on Gross Domestic Product (GDP) growth. Yet, the study emphasizes that it remains uncertain how these productivity gains would be distributed and the potential implications on work dynamics.

The study by Nguyen-Duc *et al.* [31] investigates the application of Generative Artificial Intelligence (GenAI) tools within the field of Software Engineering, emphasizing both their potential benefits and inherent challenges. The research delineates 78 open research questions across 11 distinct areas of Software Engineering, which include Requirements Engineering, Software Design, Software Implementation, Quality Assurance, Software Maintenance and Evolution, Software Processes and Tools, Engineering Management, Professional Competencies, Software Engineering Education, Macro Aspects, and Fundamental Concerns. In the domain of Requirements Engineering, GenAI is identified as a tool for the elicitation, specification, validation, and prediction of requirement changes. Within Software Design, GenAI can facilitate the selection of design and architectural patterns, support decision-making processes, and automate continuous improvements. For Software Implementation, GenAI can be integrated into programming tasks such as code generation, code completion, API search, and the validation of generated code. In the area of Quality Assurance, GenAI offers the potential to automate test case creation, maintain tests, and integrate seamlessly into quality assurance pipelines. Regarding Software Maintenance and Evolution, GenAI can assist in code translation, refactoring, automatic program repair, and anomaly detection. In Software Processes

and Tools, GenAI can be embedded into development environments, enhance CI/CD practices, and automate build and deployment processes. In Engineering Management, GenAI's influence extends to project management, human resources, human-AI collaboration, and economic impact. For Professional Competencies, GenAI can address skill shortages, enhance productivity, and serve as an advisor for software professionals. In Software Engineering Education, GenAI can be incorporated into curricula, personalize learning experiences, and foster the development of critical skills. On a broader scale, GenAI can affect Macro Aspects such as business models, market competitiveness, intellectual property, sustainability, and ethical considerations. The research also addresses Fundamental Concerns, including industry-level evaluation, reliability, explainability, and the sustainability of GenAI. The paper references tools such as ChatGPT, GitHub Copilot, and Amazon CodeWhisperer, which are utilized for tasks in requirements engineering, software design, and implementation. Two international events significantly contributed to this research: the first international workshop on AI-assisted Agile Software Development, which examined the benefits and challenges of AI-assisted Agile development, and the international workshop on Requirements Engineering for Software Startups and Emerging Technologies (RESET), which focused on GenAI and Requirements Engineering. These events were instrumental in identifying practical challenges and future expectations for GenAI's role in software development. The paper concludes that while GenAI is poised to bring substantial changes to Software Engineering, the research in this area is still nascent, with much left to explore.

The research of Jin *et al.* [19] provides significant insights into the interaction between programmers and ChatGPT in code generation. Initially, it was observed that programmers predominantly use ChatGPT to enhance existing code rather than to generate new code. In terms of quality, a substantial portion of the generated code was deemed unusable or of low quality, with 32.8% of the snippets not being utilized. Regarding interaction, programmers who provided specific snippets achieved results more quickly with fewer clarifications, indicating that the nature of interactions affects efficiency. The utilization of code was categorized into various labels, such as "Exact Match", "Modified Code", "Documentation" and "Supplementary Information", to ensure accuracy, each code snippet was reviewed by two co-authors independently. Any disagreements were resolved through discussion until a consensus was reached. To improve LLMs like ChatGPT, the authors suggest enhancing the accuracy of code generation, addressing quality issues with more diverse and high-quality data, better understanding the dynamics of interactions, incorporating developer feedback mechanisms, and expanding use cases to include activities like debugging and code review. The dataset used, named DevGPT, was created by collecting ChatGPT interactions from GitHub and comprises 17,913 prompts and responses, including 11,751 code snippets, collected between July and August 2023. The analysis included manual data cleaning and labeling to ensure relevance. The study

posed two research questions: Research Question 1 (RQ1) investigates how programmers interact with ChatGPT, identifying at which stage of development it is used, the duration of conversations, and the request strategies employed. Research Question 2 (RQ2) evaluates the utility of the generated code, especially in Pull Requests, categorizing code usage and examining how it is integrated into projects. The results revealed significant insights into how developers interact with the tool and the effectiveness of the generated code. Regarding the Generated Code Usage Rate, it was found that 16.8% of conversations resulted in code used exactly as generated, without modifications, and 26% of the generated code was used after significant modifications, such as variable renaming or adding exception handling. A considerable portion of the generated code was not used directly but served as supplementary information or inspiration. Most interactions (64%) focused on improving existing code, indicating that developers frequently use ChatGPT to refine and optimize existing code rather than generate entirely new code. Conversations were categorized into different types of requests, with a predominance of improvement requests. The study concludes that ChatGPT can be a valuable tool for developers, especially in improving existing code. The ability to generate code that is frequently used, even if modified, suggests that language models play a significant role in supporting software development.

2.7 Security Implications of Using LLMs

The study conducted by Asera *et al.* [4] presents a comparative empirical analysis of code generated by the Copilot tool, with a focus on security issues. The research aimed to determine whether Copilot introduces vulnerabilities into software as frequently as human developers do. The study utilized samples from the Big-Vul dataset, which comprises vulnerabilities in C and C++ code projects from 2002 to 2019. A total of 4,432 samples were analyzed to identify the presence of vulnerabilities and their fixes. During the analysis, the suggestions generated by Copilot were categorized based on whether they reintroduced the vulnerability or matched the original fix, using a two-stage process. The categories included exact matches with the original vulnerable code, matches with the fixed code, and other categories for suggestions that did not fit the previous ones. The results of the empirical analysis concluded that Copilot is not as bad as human developers at introducing vulnerabilities in code. It was found that in approximately 33% of cases, Copilot replicated the original vulnerable code, and in 25% of cases, it replicated the fixed code. Additionally, variations in Copilot's performance were identified across different types of vulnerabilities, with a tendency to generate vulnerable code in response

to prompts corresponding to older vulnerabilities. The study also concluded that using Copilot to fix security bugs is risky, as the tool introduced vulnerabilities in one-third of the analyzed cases. Finally, the study indicates that further investigation into Copilot's behavior is hindered by the lack of access to the training data used in its programming.

The research conducted by Pearce *et al.* [33] highlights a growing interest in utilizing artificial intelligence (AI) systems to assist in the design of computational systems, including tools that automatically generate code, such as GitHub Copilot. However, the study points out that Copilot has processed a vast amount of unverified code, which may contain flaws, potentially leading to exploitable and buggy code. Based on this evidence, the researchers systematically investigate the prevalence and conditions that might cause GitHub Copilot to recommend insecure code. The study describes an analysis where Copilot was tasked with generating code in scenarios related to high-risk cybersecurity vulnerabilities, such as those listed in the MITRE's "Top 25" Common Weakness Enumeration (CWE). The research evaluated Copilot's performance based on three distinct axes of code generation: diversity of weaknesses, diversity of prompts, and diversity of domains, across a total of 89 different scenarios. In these scenarios, Copilot generated 1,689 programs, of which approximately 40% were identified as vulnerable. The researchers discuss that from a security perspective, Copilot's responses are mixed, with a significant number of vulnerabilities generated (39.33% of the best options and 40.73% of the total options were vulnerable). They also argue that the variable security may be due to community code on GitHub, where more visible bugs tend to be reproduced by Copilot. Additionally, software security practices can evolve over time, causing approaches once considered secure to become obsolete, which may lead Copilot to generate code based on outdated methods. The research identifies several limitations in evaluating the security of the generated code, including limitations of the CodeQL tool, issues with statistical validity, and the challenge of reproducing results generated by a generative model. The study provides new insights into Copilot's behavior in security-relevant scenarios, and future work should explore other avenues, including adversarial approaches for enhanced security training. The research concludes that while tools like GitHub Copilot can boost software developers' productivity, it is crucial for developers to remain vigilant when using it, pairing it with appropriate security tools both during training and generation to minimize the risk of introducing vulnerabilities.

2.8 Other Applications of LLMs in Software Engineering

The use of large language models (LLMs) in software engineering has grown significantly, prompting Hou *et al.* [17] to conduct a systematic literature review on their applications, capabilities, and challenges. This review synthesizes the landscape of LLM-driven solutions in software engineering, from code generation to bug detection, and identifies key areas where these models are transforming traditional workflows. The authors examine various LLMs, including GPT and CodeBERT, to highlight their contributions and limitations in software development contexts. Hou et al. identify several prominent applications of LLMs in software engineering, such as code synthesis, where models automatically generate code snippets based on user input, and automated testing, where LLMs assist in creating test cases. They also discuss LLMs' role in software maintenance and documentation, which allows developers to streamline repetitive tasks and focus on more complex problem-solving. Despite these advantages, the review notes substantial challenges, such as LLMs' occasional generation of syntactically correct but semantically flawed code and the risk of amplifying existing biases in training data. The review also emphasizes the importance of tailored evaluation metrics for LLMs in software engineering. Hou et al. argue that traditional NLP metrics may not fully capture the effectiveness of LLMs in code-related tasks, advocating for domain-specific metrics that evaluate the functional correctness and efficiency of generated code. They further highlight the need for robust error-handling mechanisms within LLM frameworks to reduce the propagation of coding errors.

Xu *et al.* [46] introduce DomBERT, a domain-oriented language model specifically adapted for Aspect-Based Sentiment Analysis (ABSA), has an indirect relationship with Software Engineering. In Software Engineering, sentiment analysis can be used to understand users' opinions approachesnd feedback about software products, identify common problems, and improve the quality of software based on user reviews. DomBERT can be applied to analyze software reviews and ratings, extracting specific aspects (such as "interface", "performance", "support") and classifying the feelings associated with these aspects (positive, negative, neutral). This can help software engineers prioritize improvements and fixes based on user feedback. The model leverages a pre-training approach fine-tuned on domain-specific data, enhancing its ability to discern sentiment associated with particular aspects of a product or service. A key contribution of DomBERT is its focused architecture, which integrates both linguistic and domain-specific knowledge to improve sentiment analysis accuracy at the aspect level. Traditional sentiment analysis models often struggle with this level of granularity, but DomBERT's domain-oriented

training enables it to capture the subtleties of sentiment tied to specific entities or features, like product quality or customer service, which are essential for applications in targeted feedback analysis.

Zhang *et al.* [47] explore the potential of pre-trained transformer models for sentiment analysis in the domain of software engineering, examining the extent to which these models can accurately interpret and classify sentiments in developer discussions, issue tracking, and code reviews. Recognizing that sentiment in software engineering is often expressed through technical language, nuanced feedback, and context-specific jargon, the authors investigate whether pre-trained models like BERT and its variants can be effectively adapted to this specialized field. A primary finding of Zhang et al. is that general-purpose transformer models can achieve moderate success in software engineering sentiment analysis when fine-tuned on domain-specific data. However, the study reveals challenges in accurately identifying sentiments due to domain-specific language and the prevalence of objective, technical statements. Traditional sentiment analysis models, which perform well in general domains, often struggle to interpret the more neutral or mixed sentiments that are typical in software engineering contexts. For example, a critical comment in a bug report may not convey negative sentiment but rather technical insight, which general models may misinterpret. Through experiments, Zhang et al. demonstrate that while transformer models are adaptable, domain-specific enhancements are necessary to capture the subtleties of sentiment in software engineering. They suggest that future research should consider training transformer models with software engineering-specific sentiment annotations to improve accuracy.

As large language models such as GPT continue to be applied across various NLP tasks, their potential in automatic translation using machine translation (MT) models has become a focal point of research. Hendy *et al.* [16] present a comprehensive evaluation of GPT models in MT, comparing their performance to traditional and neural MT models across diverse language pairs. The study assesses the effectiveness of GPT models, particularly in handling low-resource languages and maintaining linguistic nuances during translation—a challenging area where traditional models often fall short. Hendy et al. highlight the strengths and weaknesses of GPT models in MT. They observe that, while GPT models perform well on high-resource languages with ample training data, they encounter difficulties with languages that have limited digital resources.

The development of pre-trained models has significantly advanced programming and natural language processing tasks, with CodeBERT by Feng *et al.* [10] representing a critical leap forward. CodeBERT is a transformer-based model that bridges natural language (NL) and programming language (PL) tasks by leveraging vast datasets of paired code and textual descriptions. Feng et al. employ a pre-training strategy that includes masked language modeling and replaced token detection, training the model to learn the syntax and semantics of programming languages alongside natural language comprehen-

sion. One of CodeBERT's key contributions lies in its dual ability to handle natural and programming languages simultaneously, making it valuable for code-related tasks such as code search, summarization, and generation. This is achieved through bimodal pre-training, which allows CodeBERT to align and understand the contextual relationship between code snippets and their NL explanations. By integrating large-scale data for both modalities, the model improves upon previous language models in handling code-based tasks but also demonstrates high adaptability across varied code languages, including Python, Java, and JavaScript. Evaluations on subsequent tasks reveal CodeBERT's efficacy, particularly in code retrieval and natural language inference (NLI) tasks, where it consistently outperforms previous models. This success illustrates the potential for pre-trained models like CodeBERT to serve as foundational tools in tasks that require the integration of natural language understanding and programming language processing.

The research conducted by White *et al.* [44] delves into software engineering techniques through the development of a set of 13 prompt patterns designed to solve various software engineering problems. These patterns are categorized into four distinct groups to assist users in efficiently navigating and applying these patterns throughout the stages from system design to implementation and maintenance. The four categories are: Requirement Elicitation, System Design and Simulation, Code Quality, and Refactoring. The focus of the study is on formulating prompt patterns that enhance code quality, refactoring, requirement elicitation, and software design. The research targets automating software engineering tasks, such as code refactoring, Domain-Specific Language (DSL) creation, API generation, and simulation of API interactions, with an emphasis on ensuring that the resulting code is independent of external libraries. It proposes the creation of strategically formulated prompts that guide the production of generic code free from specific dependencies. Supported by Large Language Models (LLMs) like ChatGPT, this methodology enables the development of API specifications from a list of requirements. The paper makes two main contributions to the field of LLMs usage in software engineering: (1) The first contribution is providing a catalog of patterns for software engineering, organized according to the type of problem they address. These patterns guide LLMs in performing software engineering tasks, such as automatically generating high-quality code, refactoring, and other activities. (2) The second contribution involves a discussion on several tested prompt patterns that facilitated improvements in requirement identification, rapid prototyping, code quality, refactoring, and system design. Thus, this research covers various stages of software development, from initial planning to code implementation, offering implementation examples and motivations for each pattern.

2.9 Final Comments

We conducted a comprehensive review of various studies related to the use of large language models (LLMs) in software engineering, with a particular focus on their application in software construction. Despite the recent advancements in LLMs, there are currently no comprehensive tools available for the complete generation of applications using these technologies. This absence raises an important question: is there a genuine need for such tools in the first place? Our approach is to first assess the practical necessity of these application-generating tools to understand the gap they might fill within the realm of software development. Based on this potential necessity, we propose our own tool, aiming to meet predefined requirements and contribute effectively to the field of software engineering.

Chapter 3

Exploratory Study

3.1 Introduction

In this chapter, we present an exploratory study involving three developers with different levels of proficiency (Subsection 3.2.2), to investigate the limitations and benefits of using ChatGPT-4 through its standard user interface. The study involves an experiment where three participants were instructed to build a web application (Subsection 3.2.3) using only ChatGPT's interface (Subsection 3.2.4), and information from a reference implementation (Subsection 3.2.1), which was built to establish the minimum requirements expected in the systems built by the participants. We detail a set of user stories and technologies that participants were required to use in building systems, including the programming language (TypeScript), front-end framework (Vue.js), and database (SQLite).

Our exploratory study aimed to understand how ChatGPT can benefit the work of developers, identify its limitations and potential improvements, and determine the situations in which its use was impaired. Throughout the study, we examined participants' interactions with ChatGPT (Subsection 3.2.5) and assessed its effectiveness by collecting feedback from developers on their use of LLMs (Section 3.3).

3.2 Methodology

The methodology of this exploratory study involved evaluating ChatGPT as a developer, using only its standard interface and without utilizing the advanced resources provided by the OpenAI API. Based on both the knowledge and prior experience of one of the participants and the pre-established requirements for the software project, a reference web system was developed manually, which served as a baseline for the tests conducted

with ChatGPT.

3.2.1 Reference Implementation

To have a reference implementation for assessing the use of ChatGPT version 4, we built a basic web-based Q&A forum named appForum, which was developed from scratch for the purpose of evaluating the capabilities of ChatGPT version 4. The rationale was to avoid evaluating an application that was possibly used by OpenAI in the training phases of ChatGPT. We use this app as a ground-truth implementation to assess and evaluate the code produced by ChatGPT. Essentially, appForum implements six user stories:

US1: As a user, I would like to register on the forum.

US2: As a user, I would like to login on the forum.

US3: As a user, I would like to create a question.

US4: As a user, I would like to delete a question.

US5: As a user, I would like to answer a question.

US6: As a user, I would like to delete an answer.

In the back-end, we defined the implementation should use TypeScript (programming language), Node.js with ExpressJS (server runtime), and SQLite (relational database). In the front-end, Vue.js and ViteJS (web frameworks) and TypeScript should be used. These technologies are widely used and well-regarded in the software development community, and they offer several advantages that make them an attractive choice for building modern web applications.

In total, the back-end implemented by the first author has 565 lines of code, three classes, and 5 files. The relational database has three tables (TB_USER, TB_ANSWER, and TB_QUESTION). The front-end has 820 lines of code, 11 files, six Vue.js components, and four pages.

Figure 3.1 shows a screenshot of the main page of our reference system.

Figure 3.2 shows a diagram with the three classes of the system (USER, QUESTION, and ANSWER) and the relationship between them.

All participants were instructed to use only the user interface from ChatGPT and to construct the application using only code generated by ChatGPT. Consequently, they were not allowed to manually create any code for any problems encountered during the construction of the application.

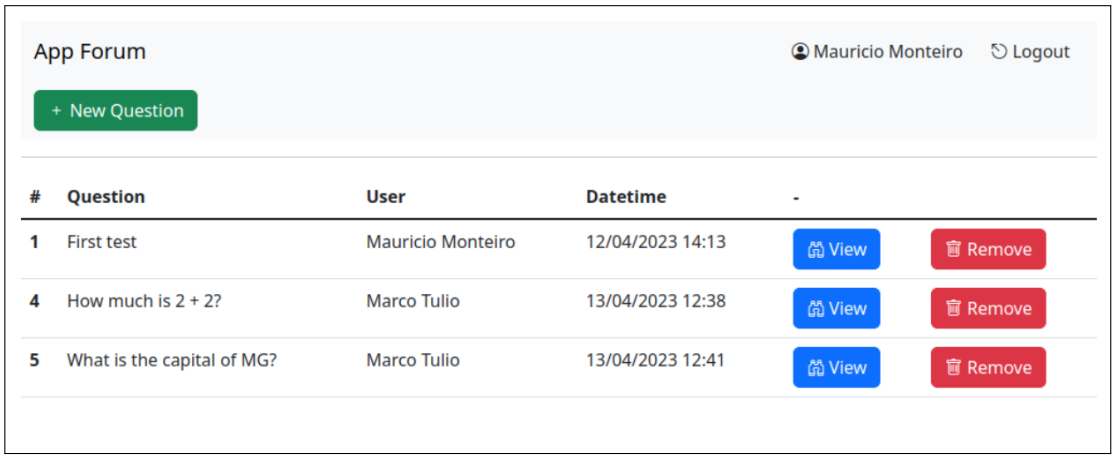

Figure 3.1: Screenshot of the main screen (reference implementation).

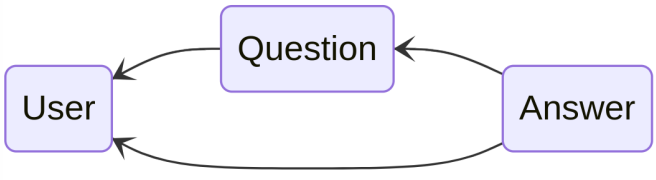

Figure 3.2: Diagram (back-end, reference implementation).

3.2.2 Participants

Three developers participated in our study, accessing our reference system, and after browsing the system, used ChatGPT to reimplement the reference web system.

The three participants in our study have differents degrees of proficiency, and the information pertaining to each participant are as follows:

Participant P1: This participant has 23 years of experience in software development. Besides being an experienced developer, he is also responsible for the reference implementation described in the subsection 3.2.1. Therefore, P1 represents the "best developer" for evaluating ChatGPT, i.e., we asked an experienced developer to use ChatGPT to generate code for an application he has previously implemented. Thus, this case—P1 having implemented the application twice, first manually and then with the support of ChatGPT—is something we deliberately chose to evaluate in the study.

Participants P2 and P3: P2 is a master student in Computer Science. He also has two years of experience in software development, and P3 is an 4th year undergraduate CS student without previous professional software development experience.

3.2.3 Inception Meeting

In the inception meeting, the participant P1 presented the features, user stories, and screenshots of our reference implementation, the appForum, to both P2 and P3. The participants were instructed to use only the user interface of ChatGPT and to rely on ChatGPT to produce code that results in an application that is as close as possible to our reference implementation and that uses the same technologies. In other words, P2 and P3 were instructed to rely only on ChatGPT to implement an identical app, even though participants P2 and P3 did not have access to the code of the reference implementation.

Therefore, we attempted to recruit a diverse set of participants in terms of their software development experience and knowledge of the application to be developed with the support of ChatGPT. It is also worthnoting that the three participants had limited experience with ChatGPT, which is expected since it is a novel technology. Particularly, they have never used ChatGPT to produce a complete software application.

3.2.4 Tools Used by the Participants

Participants used only a simple text editor to copy the code generated by the OpenAI tool. Interaction was exclusively through prompts, without changing parameters such as temperature[1], maximum token limit, or other ChatGPT settings. Essentially, we tried to reproduce the same experience that non-experts have with language models when using the standard ChatGPT interface.

3.2.5 Review Meeting

After using ChatGPT separately to implement our reference system, the three participants had a series of meetings to discuss and assess the results achieved using the AI tool. In these meetings, they presented the prompts used to interact with ChatGPT

[1]In the context of LLMs, the temperature is a parameter that controls the level of randomness in the model's responses. Higher values (e.g., 1.0 or above) result in more diverse and creative outputs, while lower values (closer to 0) make the responses more deterministic and conservative. This happens because temperature affects the probability distribution of the next words chosen by the model, adjusting the entropy of text generation.

as well as executed and discussed the code generated by the tool. During these meetings, they also come up with the following classification for the prompts used in the study:

- Initial Prompts: prompts that describe key functional and non-functional requirements, as well as prompts for configuring the project and installing the necessary frameworks.
- Feature Prompts: prompts requesting the implementation of the features of the project, including prompts that request behaviors that were not supported by the generated code.
- Bug-fixing Prompts: prompts to fix bugs or incorrect behaviors in the generated code.
- Layout Prompts: prompts to style front-end elements such as buttons, tables, and text boxes.
- Other Prompts: prompts that do not fit into the previous categories, such as requesting adjustments of configurations in the developing environment.

3.3 Experiment Results

In this section, we describe the apps constructed by each participant. First, the prompts they used are summarized in Table 3.1.

Category	**P1**	**P2**	**P3**
Initial	2	1	3
Features	26	17	13
Bug Fixing	28	24	24
Layout	7	9	0
Other	2	2	6
Total	**65**	**53**	**46**

Table 3.1: Prompts used by each participant.

Participant #1: This first participant started describing the user stories and technologies adopted in the app, using the prompt:

Despite listing the user stories in the initial prompt, P1 had to use 26 other prompts to request refinements in the code generated by ChatGPT, such as in the following prompt:

Prompt #1.1 : *I need to build a Web application in TypeScript with the following programming technologies: vue@latest, Express version 4, and SQLlite3 database. The back-end should follow a stateful architecture (e.g., user ids should be stored in sessions). The front-end should use the Bootstrap library. The app is a simple question and answer forum, which should implement the following user stories:*
** As a user, I would like to register on the forum.*
** As a user, I would like to login on the forum.*
** As a user, I would like to create a question.*
** As a user, I would like to delete a question.*
** As a user, I would like to answer a question.*
** As a user, I would like to delete an answer.*

Prompt #1.2 : *Please, create a new page such that the user can view the answers for a selected question. In the main page, add a "View Answers" button that will then present this new page.*

As another example, P1 had to elaborate prompts explicitly requesting the record of new answers to a particular question. He also used other prompts to implement missing user stories and to include missing information, e.g., the name of the user who answered each question.

Prompt #1.3 : *In the Answers Preview Screen, we should be able to register an answer to a question.*

Moreover, P1 used 28 prompts to fix bugs in the code produced by ChatGPT, such as:

Prompt #1.4 : *There is an error in the browser console: Uncaught SyntaxError: ambiguous indirect export: 'setAuthenticated'. Could you fix it?*

Finally, P1 used seven layout prompts. For example, in the code initially produced by ChatGPT the data about questions (ID, title, etc) was presented in the page as items of a list and not as rows of a table, as planned by the participant. Thus, P1 used the following prompt to request the correct layout:

Prompt #1.5 : *I would like the 'client/src/views/Questions.Vue' file responsible for registering questions to list the questions in a table with the question ID, the question title and the commands to view the answers to the question and delete the question.*

Figure 3.3 shows a screenshot of the app implemented by P1 using ChatGPT. The page presented in this figure is used to post answers.

Figure 3.3: Screenshot of the answers screen (Participant P1).

Participant #2: In this case, only one initial prompt was needed, as follows:

> **Prompt #2.1** : *I would like to create an user authentication form web page, with the fields "e-mail" and "password", with two green buttons: "sign up", which will redirect the users to a new form page where he can sign up, and "sign in", which will authenticate existing users and redirect them to a home page, returning an error when the user is not registered. I want to use the following technologies for this: Vue.js, Express.js, TypeScript and Sqlite. Could you help me with that, from installing these technologies to build those pages.*

As the reader may notice, P2 started by requesting the implementation of an specific feature (authentication), by providing a high level text describing the main fields and buttons, complemented with a request for basic error handling when the user does not exist, and then explicitly listing the desired technologies for the project. In the end, he asked ChatGPT to help in the whole process, from installing the required technologies to build the specified page.

As a response to this initial prompt, ChatGPT recommended to divide the problem in smaller steps and provided guidelines to each one (e.g., separating code by domain and indicating where newer code from an existing domain should be added). After everything was configured, ChatGPT suggested the code for the sign up/sign in feature.

However, P2 figured out that ChatGPT's answer was not fully correct and functional, which required him to write another 17 prompts to fix bugs in the generated code. As examples of such bugs, we can mention libraries that were actually not installed or not imported where needed, the back-end code was not correctly integrated with the front-end, and there were missing parameters on the project's configuration files.

For the remaining features, P2 changed his strategy and asked ChatGPT to first generate the front-end code with mocked objects. This first version of the front-end was

then carefully tested. After that, it was integrated with the back-end code. For this last step, P2 decided to use prompts that include both the front-end and the current back-end code. Such prompts requested ChatGPT to extend the back-end with logic to handle the new features that were previously implemented and tested in the front-end. As an example of this new strategy, we have the following prompt, where P2 requests ChatGPT to generate code in the back-end (file APP.TS) to persist an answer available in a form in the front-end (file POSTDETAILS.VUE).

Prompt #2.2 : *Now, I want to get the post's answers available in PostDetails.vue and, when submitting the form with the answer, I want it to be saved in the database. My code from PostDetails.vue looks like that: [Source code from PostDetails.vue file] and my app.ts looks like that: [Source code from app.ts file]*
How can I do that?

Regarding the style prompts, P2 was able to obtain the desired style by describing how the elements should look like, for example by informing the hexadecimal code of particular elements' colors or the shape of buttons (e.g., a button with a stadium-shaped border). He also experienced naming the colors and asking for darker or clearer tones of existing ones, which ChatGPT understood as well.

Figure 3.4 shows a screenshot of the page to visualize questions and to provide answers, as implemented by P2 using ChatGPT.

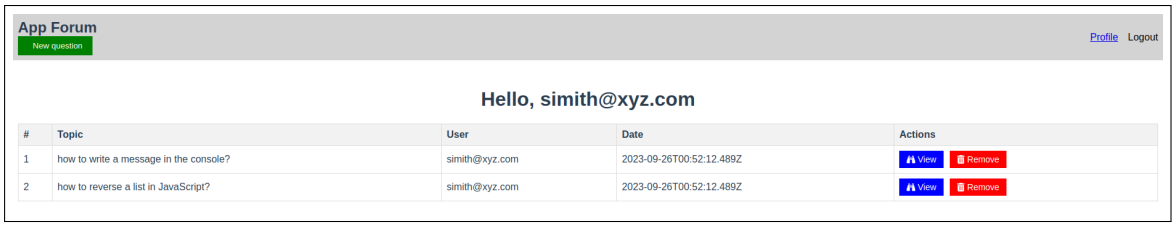

Figure 3.4: Screenshot of the question list page (Participant P2).

Participant #3: This participant started with the following prompt that describes the technologies and basic features of the application and asks ChatGPT to build the code:

Prompt #3.1 : *Please, build a forum app with the following technologies: Vue.js, Express.js, TypeScript, and Sqlite. The app should have a login screen and an option to register a user if he/she haven't already. After logging in, the user should have the option to create a Question and to see the questions that have been created before. By clicking on the "Answer" button, a new screen will appear and the user will be able to provide an answer, save it, and then return to the main screen.*

The answer of this first prompt included a good portion of the project files and also instructions to install the required technologies. However, two more prompts were necessary to obtain details about such instructions, including the following prompt:

Prompt #3.2 : *In topic 5 ("Set up the front-end"), how do I create views for each route?*

Prompt #3.3 : *Please implement code to make API calls to the back-end server for user registration, fetching questions, and saving new questions.*

After these initial prompts, P3 asked ChatGPT to implement other features by using 13 prompts, such as:

However, the code generated by ChatGPT missed some important files, such as index.js (in the backend). Indeed, ChatGPT informed that it was necessary to implement this file, but it does not provided the code, even after an explicit request. As a result, several bugs persisted in the generated code. Moreover, important user stories such as user registration and login were not properly implemented. The front-end for such features was created but it was not able to call the correspondent code in the back-end. ChatGPT correctly attributed this problem to an error in the connection between the front-end and back-end. However, when P3 attempted to fix the bug, ChatGPT entered in a loop, continuously suggesting previous (and also incorrect) versions of the code. After 18 attempts, P3 concluded that it was not possible to advance and he decided to quit with the project not finished.

3.4 Lesson Learned

The main lesson learned from this study is the importance of the developer's proficiency in the technologies and frameworks used by the target system. For instance, the first two participants—who successfully completed the proposed application—were well-versed in web development and its associated technologies and frameworks. As a result, they were able to leverage their experience to formulate prompts that guided ChatGPT in fixing the bugs in the code generated by the tool.

Lesson Learned #1 : When using the standard interface provided by ChatGPT to support the end-to-end construction of web apps, experience in software development practices, architectures, and technologies is crucial. This expertise is especially important when formulating bug-fixing prompts. In other words, it is unrealistic to expect non-developers to write these prompts and create a functional web app using ChatGPT.

3.5 Threats to Validity

There are two main threats to the validity of the results reported in this exploratory study. First, our reference application (a question and answer forum) may not represent the universe of systems that are built from scratch by software developers. However, we chose a well-known application that follows a common architecture and that uses popular technologies. Second, the code of the reference application was generated by ChatGPT with prompts formulated by three developers. Therefore, these participants may not represent the universe of developers who intend to use ChatGPT to support end-to-end software construction. However, we selected developers with diverse profiles and levels of experience in software development.

3.6 Analysis Using Source Code Metrics

In this section, we describe an analysis of the code generated by ChatGPT. For this purpose, we use metrics to analyze the size and complexity of this code. We will use three versions of the code: the oracle (manually generated without ChatGPT support) and the versions generated from the prompts proposed by participants P1 and P2 (both with code generated entirely by ChatGPT). We will not analyze the version by participant P3 because it was not successfully completed.

To conduct this analysis, we decided to use three size metrics (number of files, number of functions, and number of lines of source code) and one complexity metric (Cyclomatic Complexity). These are well-known metrics, and we found them sufficient to uncover many aspects and characteristics of the generated code. Therefore, we opted not to consider other metrics, such as those related to cohesion and coupling, for example.

3.6.1 Number of Files

Table 3.2 shows the results for number of files. Analyzing the results, we can observe that there are no significant differences between the three versions of the code. They have, respectively, 16 files (oracle and P1) and 14 files (P2). However, if we analyze

by architecture component, we begin to notice an important difference in the number of files, especially in the back-end. This number is equal to five files in the oracle, seven files for the version generated by P1, and only two files in the version generated by P2. We further analyze these differences in more detail.

Table 3.2: Number of files.

	Oracle	P1	P2
Front-End	11	9	12
Back-End	5	7	2
Total	**16**	**16**	**14**

In the oracle, the back-end code, as recommended by good software architecture practices, was divided into five files responsible for: (1) application initialization and database connection (main.ts); (2) instance control and database manipulation (database.ts); (3) ExpressJS framework configuration (app-web.ts); (4) publishing and implementing the endpoints of the back-end services (end-point.ts); and (5) implementation of the business rules responsible for the inclusion and exclusion of the key entities of the system (entity.ts). Specifically, three entities were created: User, Question, and Answer.

In version P1, the code generated by ChatGPT was organized into seven files responsible for: (1) initializing the database connection (db.ts); (2) initializing the entire application (index.ts); (3) user authentication (middlewares/authenticate.ts); (4) the user registration endpoint (routes/auth.ts); (5) the endpoint to handle forum questions (routes/questions.ts); (6) the endpoint to handle forum answers (routes/answers.ts); and (7) an interface that assists in authentication (types/RequestWithUser.ts).

In version P2 , ChatGPT implemented all these requirements in only two files: (1) Implementation of all endpoints, as well as business rules and configuration of ExpressJS and database connection (app.ts); and (2) File with an interface with user attributes (types.ts).

The following three diagrams help to gain a better understanding of the oracle architecture (Figure 3.5), as well as the code generated by ChatGPT in versions P1 (Figure 3.6) and P2 (Figure 3.7). Comparing the oracle architecture with the one of P1, we can notice that ChatGPT did not partition the app into files responsible for endpoints, business logic, and database access. Instead, the partitioning was by functionality, i.e., user (routes/auth.ts), question (routes/questions.ts), and answer management (routes/answers.ts). Each of these files monolithically contains the code for endpoints, business logic, and database access. Thus, this is a different architectural organization from that adopted in the oracle and also from the usual architecture adopted in web applications.

Comparing the oracle architecture with the one of P2, we can see that the P2 essentially is a monolithic application that implements all requirements (endpoints, busi-

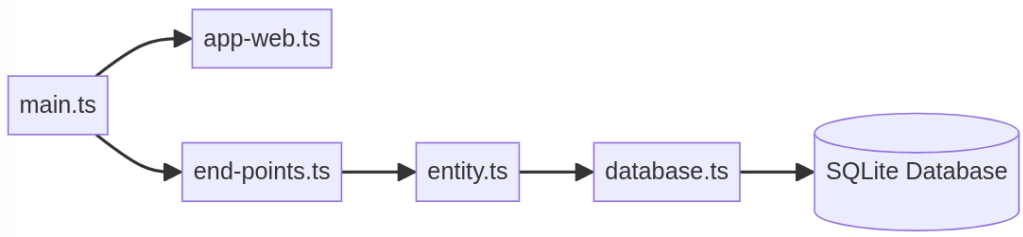

Figure 3.5: Architecture of the oracle version (with a clear separation between web end-points, business rules, and database persistence).

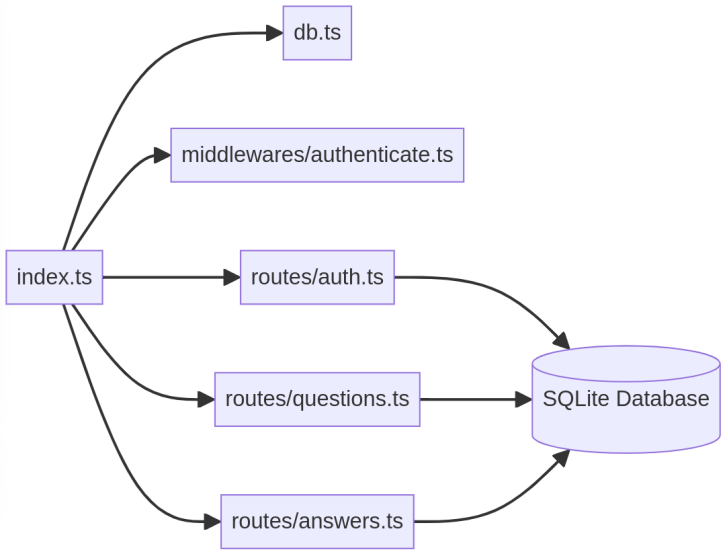

Figure 3.6: Architecture of the P1 version, which follows a functional decomposition, i.e, we have modules responsible for each feature, which implement both non-functional (e.g., web end-points) and functional concerns.

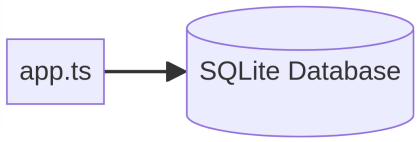

Figure 3.7: Architecture of the P2 version. This is a monolithic architecture where everything is implemented in a single file.

ness logic, and database access) in a single file (app.ts). Especially in the case of a large system, this solution is not recommended and can increase system maintenance costs.

Lesson Learned #2 : When using ChatGPT for end-to-end generation of applications, it is important to define or at least check the software architecture of the generated code. Failing to do so may result in the generation of a monolithic architecture, where all requirements, both functional and non-functional, are implemented in a single file. This occurred, for instance, with the code generated by the second participant of our study.

3.6.2 Number of Functions

Table 3.3 shows the number of functions in the three versions of the code we analyzed. We can observe that the oracle has a significantly higher number of functions (64 functions) compared to versions P1 and P2 (30 and 35 functions, respectively). This difference can be attributed to the distinct architectural approaches adopted in these versions, as discussed in the previous section. For instance, in the oracle, separate functions are dedicated to configuring endpoints, implementing business rules, and persisting data in the database. In contrast, in versions P1 and P2, these responsibilities are implemented monolithically within a single function.

Table 3.3: Number of functions.

	Oracle	P1	P2
Front-End	19	18	13
Back-End	45	12	22
Total	**64**	**30**	**35**

3.6.3 Lines of Code

Table 3.4 shows the results for lines of code (LOC). Interestingly, P1 and P2 have 34% and 27% fewer lines of code, respectively, when compared to the oracle. To explain this difference, we conducted a more in-depth analysis, divided between back-end and front-end.

Table 3.4: Number of lines of code.

	Oracle	P1	P2
Front-End	820	581	852
Back-End	565	326	154
Total	**1,385**	**907**	**1,006**

In the back-end, both P1 (326 lines) and P2 (154 lines) exhibit fewer lines of code than the oracle (565 lines). This reduction is expected, particularly in the context of the P2 project. As discussed earlier, all the code in P2 was essentially implemented in

a single file, leading to a decrease in the overall number of lines of code. However, this approach yields a less scalable architecture and poses challenges for maintenance, as we have previously discussed.

On the front-end, however, the results are not aligned. That is, P1 (581 lines) has fewer lines of code than the oracle (820 lines), while P2 (852 lines) has more lines of code. This difference is related to the fact that P1 generally worked without focusing on screen details and only focusing on ensuring that the user stories were functionally implemented, even if visually the application differed from the oracle. On the other hand, P2 used an approach that first aimed to replicate the front-end layout as faithfully as possible. To illustrate, we show three screenshots of the page with user information. We can observe that P2 (Figure 3.10) implemented a separate page to display this information, as in the oracle (Figure 3.8). On the other hand, P1 (Figure 3.9) chose to show this information on the questions and answers page. Thus, a new page was not needed for this purpose.

Figure 3.8: User information page (Oracle).

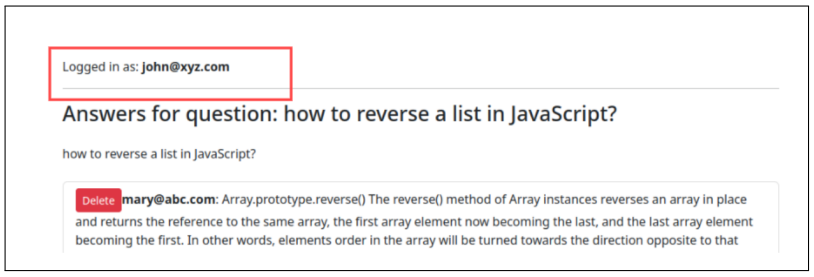

Figure 3.9: Answers page, with information about the user in the top (Participant P1).

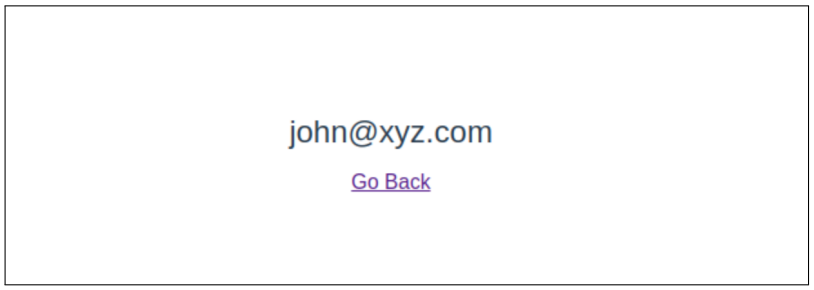

Figure 3.10: User information page (Participant P2).

Lesson Learned #3 : When using ChatGPT for end-to-end application generation, it is important to specify, in details, the desired pages and components of the application, as demonstrated by P2. Otherwise, the interface layout may end up being overly simplistic and distant from an elaborated and professional version.

3.6.4 Cyclomatic Complexity

We will also segment this analysis into back-end and front-end components. First, Figure 3.11 show histograms with the Cyclomatic Complexity (CC) values of the back-end functions in the analysed projects. These values were computed using the JSHint tool.[2] As we can see, most back-end functions have a low cyclomatic complexity: in the oracle, 88% of the functions have complexity less or equal two; in versions P1 and P2, this percentage is 66% and 90%, respectively.

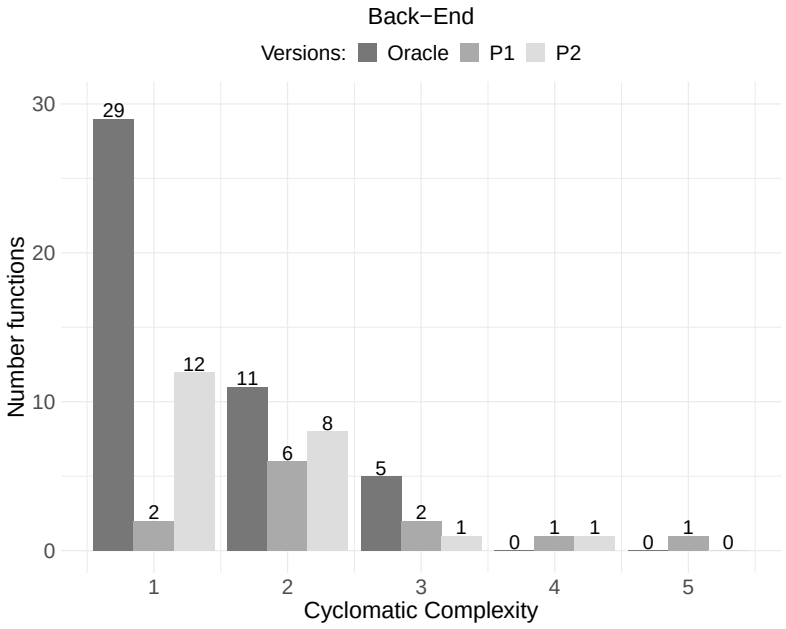

Figure 3.11: Histogram with Cyclomatic Complexity values (back-end).

In the three versions (oracle, P1, and P2), the back-end function with the highest CC is responsible for user authentication. Listing 3.1 shows the implementation of this function in version P1, with a CC value of 5. In this code, there is an IF statement that checks two different variables, as well as include a conditional assignment (lines 7-9). Thus, we can generally conclude that this function does not have a complex logic,

[2] `https://jshint.com`

especially if we assume that this is the most intricate function among the analyzed back-end implementations.

```
router.post('/login', async (req, res) => {
  try {
    const db = await dbPromise;
    const {email, password} = req.body;
    const user = await db.get('SELECT * FROM users WHERE email = ?', [email]);
    if (user && (await bcrypt.compare(password, user.password))) {
      const secretKey = process.env.JWT_SECRET || 't2bO36uAY6A19yoRyvnOd6yH';
      const token = jwt.sign({id: user.id}, secretKey, {
        expiresIn: '1h', // Set the expiration time for the token, for example,
            1 hour
      });
      res.status(200).json({message: 'Logged in successfully', user, token});
    }
    else {
      res.status(401).json({error'Invalid email or password'});
    }
  }
  catch (err) {
    console.error(err);
    res.status(500).json({error: 'Failed to log in'});
  }
});
```

Listing 3.1: Authentication function in the version of P1

Regarding the front-end, the functions are usually less complex than the back-end ones, as can be checked in the histogram presented in Figure 3.12. In the oracle, 89% of the front-end functions have CC ≤ 2; in versions P1 and P2, this percentage is 94% and 100%, respectively. The maximum CC is equal to 3, and it is observed in the oracle (in three functions) and in P1 (in five functions).

Lesson Learned #4 : Overall, the functions generated by ChatGPT exhibit low cyclomatic complexity. In our dataset, considering all functions from P1 and P2 (back-end and front-end), 89% have a CC ≤ 2. This implies that comprehending and maintaining the generated code is not likely to be challenging.

3.7 Static Analysis Tools Evaluation

In this section, we describe the analysis of the code generated by ChatGPT using static analysis tools. To this end, we used two widely recognized and consolidated tools:

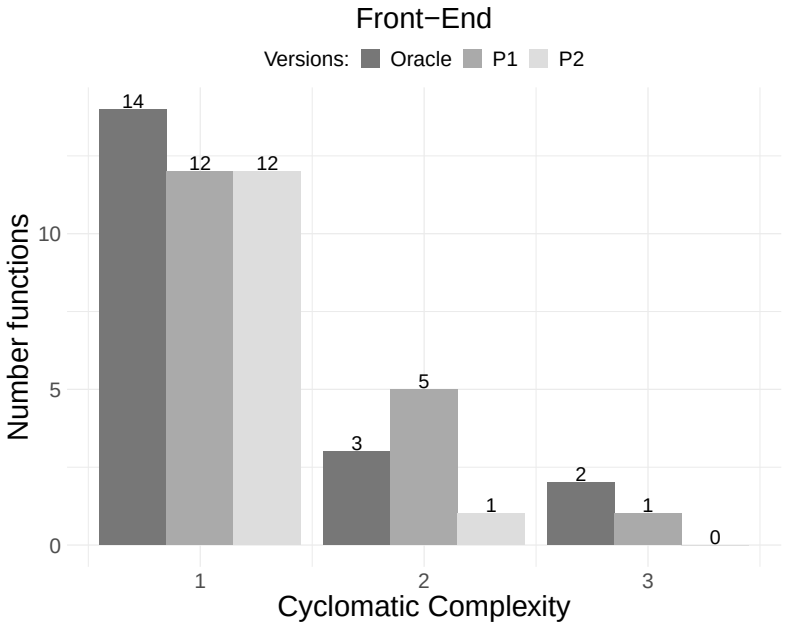

Figure 3.12: Histogram with Cyclomatic Complexity measures (front-end).

ESLint[3] (an open-source tool with 24K stars on GitHub) and SonarQube[4] (a commercial tool used by more than 400K worldwide organizations, according to the tool's site). Our analysis will be carried out on the oracle, P1, and P2 versions. For the same reason as explained in Section 3.6, we will not analyze the version generated by participant P3.

In fact, the execution of both tools in the oracle did not report any issues. Thus, we only report results for P1 and P2 in the following tables.

3.7.1 ESLint Results

Table 3.5 shows the issues encountered by the ESLint tool (executed using its default configuration for both back-end and front-end components). In P1, the tool reported five issues. Four were classified as @vue/multi-word-component-names, and one as @typescript-eslint/no-unused-vars. In P2, ESLint reported six issues: three were classified as @typescript-eslint/no-explicit-any, and the other three as @typescript-eslint/no-unused-vars.

According to the ESLint documentation, @vue/multi-word-component-names refers to components whose names are not composed of several words, which is useful to avoid conflicts with existing or future HTML elements. Such elements always have a single word. For example, a component with the name todo, instead of todo-item. While

[3] `https://eslint.org`
[4] `https://www.sonarsource.com/products/sonarqube`

Table 3.5: Issues by ESLint.

Issues	P1	P2	Total
@VUE/MULTI-WORD-COMPONENT-NAMES	4	0	**4**
@TYPESCRIPT-ESLINT/NO-UNUSED-VARS	1	3	**4**
@TYPESCRIPT-ESLINT/NO-EXPLICIT-ANY	0	3	**3**
Total	**5**	**6**	**11**

TODO is not a currently an HTML tag, it is prudent to refrain from using it to prevent potential conflicts in case future versions of HTML decide to introduce TODO as a new tag.

The @TYPESCRIPT-ESLINT/NO-UNUSED-VARS issues refers to variables or imports that are not being used. Finally, the @TYPESCRIPT-ESLINT/NO-EXPLICIT-ANY issues refers to variables without a defined type. An example of this issue can be seen in Listing 3.2 where the array ROWS accepts any type of elements (line 2).

```
1 app.get('/api/posts', (req, res) => {
2  db.all('SELECT * FROM posts', (err: Error, rows: Array<any>) => {
3    if (err) {
4      res.status(500).send({ error: 'An error occurred while fetching posts.' });
5    } else {
6      res.send(rows);
7    }
8  });
9 });
```

Listing 3.2: Example of @TYPESCRIPT-ESLINT/NO-EXPLICIT-ANY in P2

3.7.2 SonarQube Results

We executed SonarQube in both P1 and P2 (back-end and front-end). In both projects, the tool reported issues in two categories only (bugs and code smells). Table 3.6 shows the bugs found by the SonarQube tool. One bug was found in P1, and two bugs were identified in P2. In the case of P1, the only identified problem is related to the lack of a description for a table to assist visually impaired users. Of the two bugs found in P2's code, one is also related to the absence of a table description. The other is associated with the use of a font style outside the system standard.

Table 3.7 shows the code smells found by SonarQube. As can be seen, the tool identified three code smells in P2. They refer to the unnecessary use of imports (2 cases) and comments (1 case).

Table 3.6: Bugs by SonarQube.

Bugs	**P1**	**P2**	**Total**
Add a description to this table	1	1	**2**
Unexpected missing generic font family	0	1	**1**
Total	**1**	**2**	**3**

Table 3.7: Code Smells by SonarQube.

Code Smells	**P1**	**P2**	**Total**
Remove this unused import of "..."	0	2	**2**
Remove this commented out code	0	1	**1**
Total	**0**	**3**	**3**

Lesson Learned #5 : The code generated by ChatGPT has good quality, as indicated by an analysis conducted using two widely used static analysis tools (ESLint and SonarQube). Overall, both tools identified few bugs, issues, and code smells. Additionally, the ones that were identified are not severe or difficult to fix.

3.8 Final Remarks

In this chapter, we presented the results of an exploratory study where we evaluated ChatGPT, using only its standard User Interface without any advanced features or temperature settings. The experiment revealed that a developer's proficiency level directly influences their ability to build web software and affects the quality of ChatGPT-generated code. Our study demonstrates that using ChatGPT solely through its interface is not beneficial for novice developers, as the generated code does not take into account the developer's environment and assumes they have the general knowledge required to correctly integrate the code into the project's files and directories. The findings emphasize the importance of tailoring AI tools to match the skill levels and specific needs of developers to enhance their software development tasks effectively.

The data related to the exploratory study is at the link: `https://github.com/mauricioms/Results-Generated-with-ChatGPT`

Chapter 4

NoCodeGPT Tool

4.1 Introduction

This chapter discusses the development of the NoCodeGPT tool as a supportive tool for building Web apps with Language Models, in response to the difficulties encountered by novice developers in our exploratory study. The development of this tool, looked to reduce the primary challenges and increase individual productivity by providing straightforward resources for software construction. It combines the principles of NoCode tools with Large Language Models (LLMs) to simplify the development process, thereby reducing the challenges faced by novice developers. The NoCodeGPT Tool is designed to aid developers, especially those with limited experience, by integrating technological knowledge and standardizations that streamline the software construction process.

In this chapter, we will discuss the main functionalities (Subsection 4.3), as well as the architecture implemented to build the tool (Subsection 4.4).

4.2 Features and Architecture

Based on the lessons learned from our exploratory study, we decided to implement a new interface for GPT, with specific features for building small web apps using only prompts (i.e., without writing any code). This interface, called NoCodeGPT, can act as a replacement for the traditional interface (ChatGPT). The idea is to enable not only developers (like participants P1 and P2 from the exploratory study) to use language models for generating web applications. Instead, we intend that users with limited software development experience (like participant P3) could also be able to generate small web applications without having to write a single line of code.

In this section, we first describe the key features of NoCodeGPT (Section 4.3).

These features emerged from our experience and observation in the Exploratory Study (Chapter 3). Next, we also present the architecture and implementation of NoCodeGPT (Section 4.4).

4.3 Main Features

The main functionalities of the tool are as follows:

Initial Prompts: The tool defines internally the initial prompts for building web systems, meaning the prompts that specify the system's technology, architecture, main directories, among other decisions. This way, the user does not need to create or provide these prompts. Additionally, NoCodeGPT requests that the user provide a prompt describing the core functionality of the system under construction (see Figure 4.1). This prompt is important for establishing the contextual framework to be considered by the GPT model when generating the system.

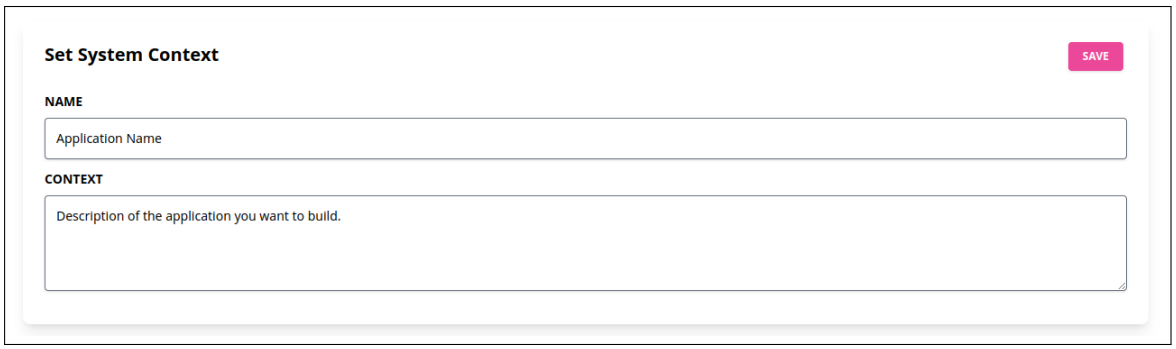

Figure 4.1: Page to enter the name and main context of the system under development.

Pre-defined features: Some features are very common in web system. Therefore, we decided to provide built-in and pre-tested prompts to automatically generate and include such features in the systems generated by NoCodeGPT. Currently, the tool supports two pre-defined features: a login and a user registration page.

Creation and Refinement of Pages: Since NoCodeGPT is exclusively designed to build web systems, it assumes that these systems consist of a set of pages. For each page, the user must provide a name and a brief description of the desired functionalities. For example, in the case of a Q&A forum, the user might create a page called *Question Submission* with the following description: "I would like to be able to submit questions on this page. For each question, I would like the system to store the following information:"

However, as we clearly learned in the Exploratory Study, it is unlikely that the GPT model will generate the ideal page the user envisions in the first iteration. Consequently, NoCodeGPT includes a refinement feature where the user can request improvements to a specific page. These prompts can be of three types: new features (such as, *the page should have a button to delete a question*); bug fixes (such as, *the delete question button is not working*), and layout adjustments (such as, *the delete question button should be positioned right after the question's title*). Figure 4.2 shows an example of this functionality for the Q&A Forum. In this figure, the user is refining the *Question Submission* page.

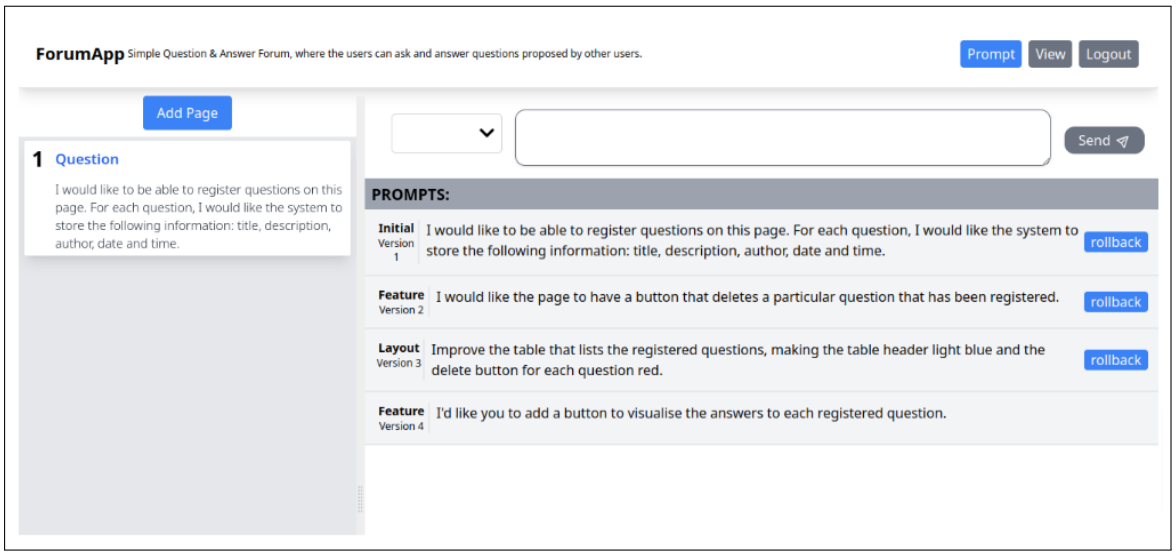

Figure 4.2: Prompts for refining generated pages.

Since refinements are restricted to one page, the tool's implementations is able to add context to the prompts provided by users. For example, our implementation automatically adds the source code files that need to be refined by the GPT model in such prompts. This functionality is essential to allow the tool to be used by inexperienced users, as they usually cannot correctly identify the source code files in which they intend to make a change. Moreover, it increases the effectiveness of language models by providing them with the precise code they need to work on.

Web Page Transitions: Another interesting aspect is a feature provided by NoCodeGPT to connect two pages. For instance, in our forum, the user must independently create the *Question Submission* and *Answer Submission* pages. However, afterward, they must return to the *Question Submission* page and request the creation of a button on each question that shows the page with its answers. An example is shown below:

Prompt: *Add a button to view the registered answers on the Answers page.*

After various tests, we concluded that this alternative using simple prompts is the best solution for connecting pages. That is, the user should first create and test the pages

individually. Then, they should request the connections between the pages, whenever necessary. To do this, they should simply use a prompt asking that when a certain component is clicked another page should be displayed.

Version Management: NoCodeGPT also implements a simplified version control system for the code generated with the assistance of GPT. This allows users to easily restore a previous version if GPT starts generating invalid code that does not comply with the system specifications. For example, suppose the user formulates prompts requesting the implementation of features $F_1, F_2, \ldots, F_n$, with each submission generating a respective version V_i of the system. However, in a particular version V_j, the language model might start hallucinating and generate invalid code. In such cases, the user can revert to version V_{j-1} using a single button provided in NoCodeGPT's interface.

This feature was specifically designed to address a difficulty encountered by Participant P3 in our Exploratory Study. This participant abandoned the study because ChatGPT chose a path that resulted in successive invalid versions of the system. Moreover, using the standard ChatGPT interface, backtracking to the last correct version was not easy. He would have had to save these versions manually, which is not a natural procedure for less experienced users.

Execution and Visualization: NoCodeGPT also provides a button that allows the user to quickly run and check the behavior of the code generated by GPT. This way, the user does not need to have knowledge of command-line tools, such as compilers and interpreters.

4.4 Architecture and Implementation

The implementation of NoCodeGPT has two modules: front-end and back-end, as shown in Figure 4.3. The front-end is responsible for controlling the functionality of the tool's pages and for guiding the user through the workflow to build the Web app. The back-end is responsible for receiving requests sent by the front-end and processing the necessary information to build the prompts that are submitted to the OpenAI API. The back-end also stores all interactions with the OpenAI platform in a database.

Specifically, the front-end implementation uses the following technologies: Vue.js (version 3) with TypeScript using the Composition API, HTML, and CSS. For the back-end implementation, ExpressJS, SQLite 3, and JSON Web Token are used. The front-end has 1,847 lines of code, and the back-end has 3,342 lines, totaling 5,189 lines.

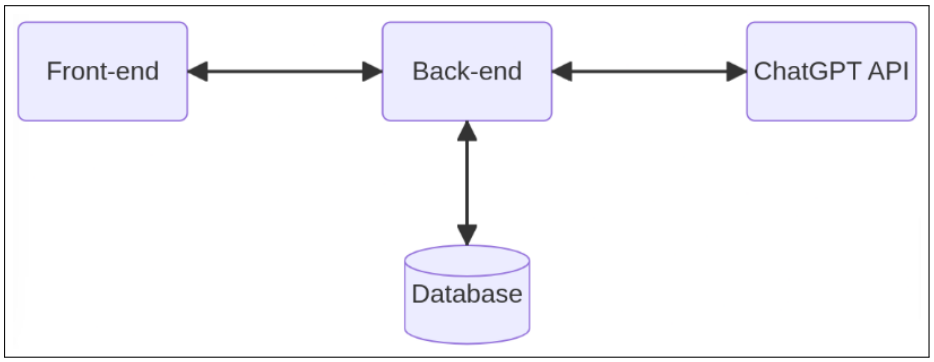

Figure 4.3: NoCodeGPT's high-level architecture.

Figure 4.4 provides a detailed view of the architecture of the implemented tool. The front-end consists of the Main class, which controls the initialization of this component, and the App class, which manages the interface logic. Page route management is handled by the Route class, which is capable of displaying any of the pages, such as AddPageView and LoginView. Specifically, the application pages are implementations of Vue Single-File Components. These components call the endpoints implemented in the back-end using the CallService class, which relies on the standardized implementation of the FetchAPI class.

In the back-end, the prompts are initially processed by adding other important information to obtain more accurate responses and, ultimately, the GPT API is called to generate the code requested by users. The PromptRequest and PromptResponse classes store, respectively, the requests and responses made to this API. Detailed information from the previous requests is also stored to implement the version control mechanism provided by the tool. Finally, the PromptType class stores the templates of the prompts sent to the OpenAI platform.

Temperature: From the experience of the Exploratory Study, we concluded that lowering the temperature in fact makes the model more deterministic and also accurate. Therefore, we decided to reduce the temperature used in NoCodeGPT to zero.

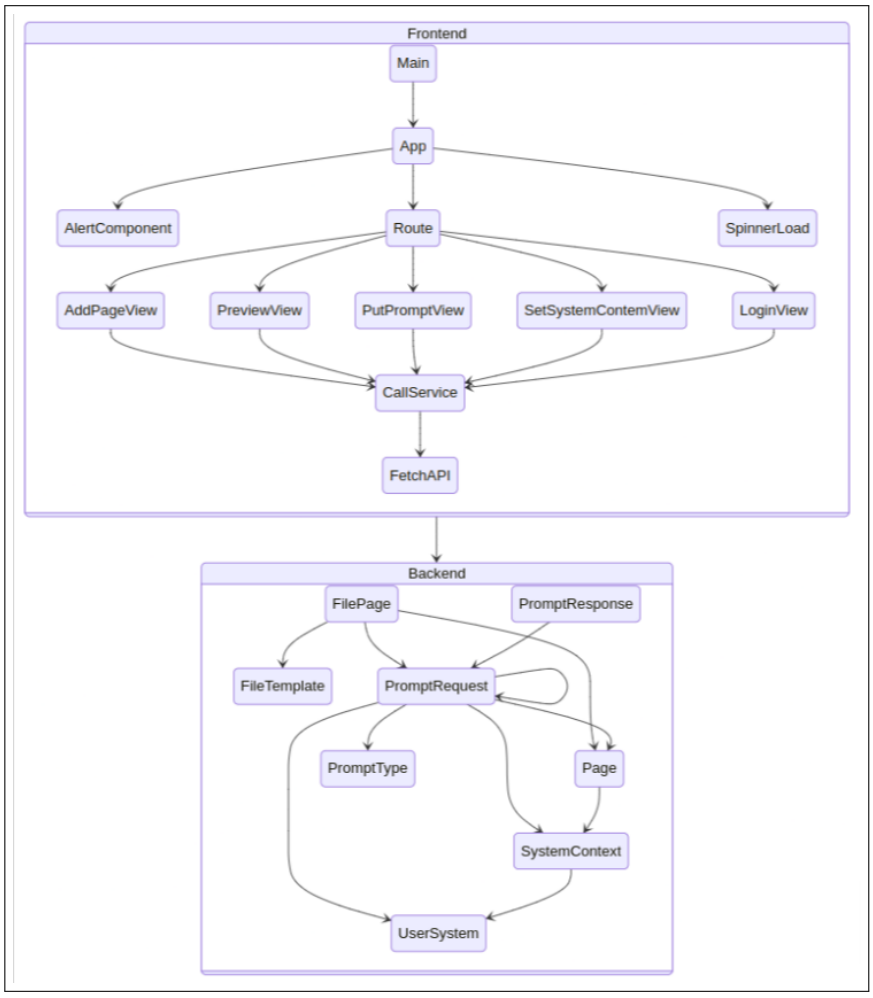

Figure 4.4: Front-end and Back-end architecture.

4.5 Final Remarks

In this chapter, we explored the development of the NoCodeGPT tool, created to mitigate the main limitations highlighted by P3 participant in our exploratory study (Chapter 3). We presented all the features of the tool, from the ability to contextualize to view an application. We emphasize the importance of temperature parameterization, a crucial factor for tool efficiency. We observed that it works as an intermediate layer between the user and the OpenAI API. The features demonstrated comprise a set of resources that aim to improve user experience and efficiency in software construction, especially for those with little coding experience. The source code of the NoCodeGPT tool can be accessed by the link: `https://github.com/mauricioms/nocodegpt`

Chapter 5

NoCodeGPT Evaluation

5.1 Introduction

In this chapter, we will discusses our evaluation, realizaded by a group of participants who share the same profile as those in the previously mentioned exploratory study (Chapter 3). The evaluation of NoCodeGPT was conducted in two phases, with a total of 14 participants, who were invited to implement two small web applications using only prompts and our tool. Our main objective is to verify whether NoCodeGPT helps to overcome the limitations identified in the Exploratory Study. The two distinct experiments were designed to generate results that can be evaluated based on applications with different characteristics. This methodological approach will facilitate a more precise analysis of the level of efficiency achieved by the tool.

5.2 Methodology

The evaluation of the proposed tool was conducted in two phases. First, a pilot test was carried out with four participants. The goal was to test and validate the implementation of the proposed tool with a small number of participants and assess its feasibility. Specifically, the participants used the tool to implement a simple task management application with the following user stories:

1. As a user, I want to add a new task to the list so I can track what I need to do.
2. As a user, I want to edit an existing task so I can correct or update the task description.
3. As a user, I want to mark a task as completed so I can indicate that I have finished it.

4. As a user, I want to remove a task from the list so I can clean up my todo list.

The participants received initial training on the tool (30 minutes) and were also provided with a low-fidelity mockup of the application. We decided to provide this mockup so that participants could have an idea of the functionality they needed to implement. However, we clarified that the mockup would serve only as an example, meaning the application to be created did not need to have an identical layout.

Following this pilot experiment, we conducted a larger experiment with 10 participants. In this case, we used the ForumApp from our Exploratory Study (Section 3) as the target application. The participants also received initial training and a mockup of the requested app. They were asked to implement the following user stories:

1. As a user, I want to add a new question to the forum.
2. As a user, I want to delete a question from the forum.
3. As a user, I want to access the answers of a given question in a separated page.
4. As a user, I want to answer to a question in the forum.
5. As a user, I want to delete an answer.

Compared to the Exploratory Study, we made two important changes to the list of user stories of ForumApp. First, we removed the stories related to login and user registration, as these are predefined features in NoCodeGPT, as explained in Section 4.3. Second, we added a story explicitly stating that, given a question, it is important to access a second page with its answers. This was done to prevent users from implementing the entire app on a single page. If that happened, we would not be able to evaluate NoCodeGPT's ability to support the creation of apps composed of multiple web pages.

Table 5.1 summarizes the participants' web development experience in each experiment (TodoApp and ForumApp). All participants are undergraduate Computer Engineering students in their first or second year. Additionally, as shown, they match the ideal user profile for NoCodeGPT, meaning they have either no experience or, at most, one year of experience in web development.

Years of experience	**TodoApp**	**ForumApp**
No experience	2	8
Less than one year's experience	2	2

Table 5.1: Participants' experience on web development.

5.3 Pilot Experiment Results (TodoApp)

In the pilot experiment, two participants managed to build the TodoApp application with all four proposed stories. However, two participants did not implement one of the stories. Figure 5.1 shows a screenshot of an application built by one of the participants who successfully completed all four stories.

Figure 5.1: Screenshot of participant P2's application.

Moreover, Table 5.2 details the stories that were implemented by the pilot participants. As we can see, participant P4 was unable to implement task editing, while participant P1 was unable to implement a feature to mark a task as completed. In both cases, the participants attempted to implement the functionality, but the code generated by GPT did not work as expected. Thus, they decided to give up without attempting again. On the other hand, the creation of new tasks, as well as their removal, were implemented by all four participants.

User Stories	**Participants who succeeded**
As a user, I want to add a new task	P1, P2, P3, P4
As a user, I want to edit an existing task	P1, P2, P3
As a user, I want to mark a task as completed	P2, P3, P4
As a user, I want to remove a task from the list	P1, P2, P3, P4

Table 5.2: Participants who implemented each user story in the pilot experiment (TodoApp).

Figure 5.2 shows the number of prompts each participant used to build the TodoApp. Participants P2 and P3 interacted the most with the tool and were the only ones to implement all user stories, each using 16 prompts. In contrast, participants P1 and P4 used nine and eleven prompts, respectively. The most common prompt type was for requesting features, except for participant P2, who used eight prompts to improve the app's layout. For P1, there was an even split between feature and bug-fix prompts, with four prompts

each. Interestingly, P1 and P4 did not use any layout prompts, indicating they were satisfied with the initial layout proposed by GPT. Lastly, all participants used exactly one initial prompt.

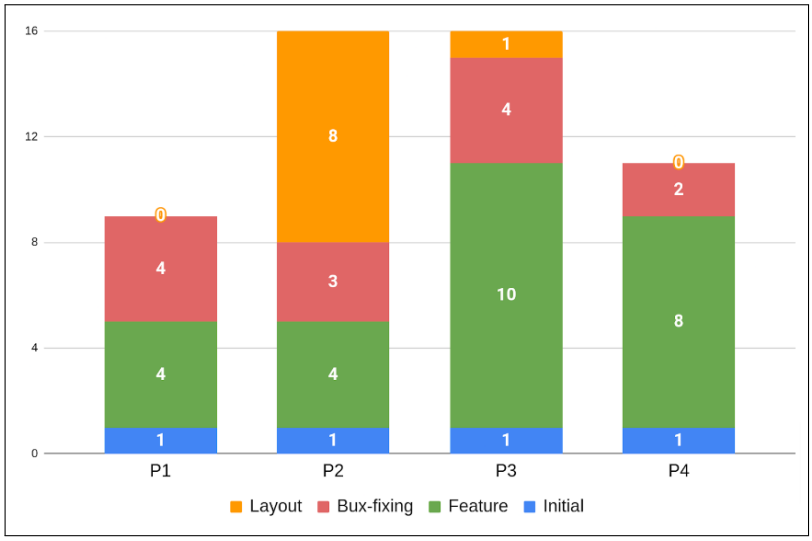

Figure 5.2: Types of prompts implemented by each participant in the pilot experiment (TodoApp).

NoCodeGPT's version control functionality was heavily used during the pilot experiment. The four participants carried out a total of 16 rollbacks while building the proposed web application. Table 5.3 indicates that participant P4 made the most use of this feature, requesting five rollbacks.

Participant	**P1**	**P2**	**P3**	**P4**
Rollbacks	4	4	3	5

Table 5.3: Rollbacks by participants in the pilot experiment (TodoApp).

Figure 5.3 illustrates the rollback operations carried out by participant P2. In this figure, each branch represents a sequence of prompts that were abandoned (i.e., that were concluded with a rollback). Specifically, P2 performed four rollbacks, after prompts 4, 6, 10, and 13. We can also see that P2 used 16 prompts in total, but seven prompts were discarded. Only nine prompts resulted in usable code (prompts 1-2, 7-9, 11, 14-16).

5.4 Results of the Second Experiment (ForumApp)

In the second experiment (ForumApp), seven participants successfully implemented all five stories. The last three participants failed to implement at least two user stories.

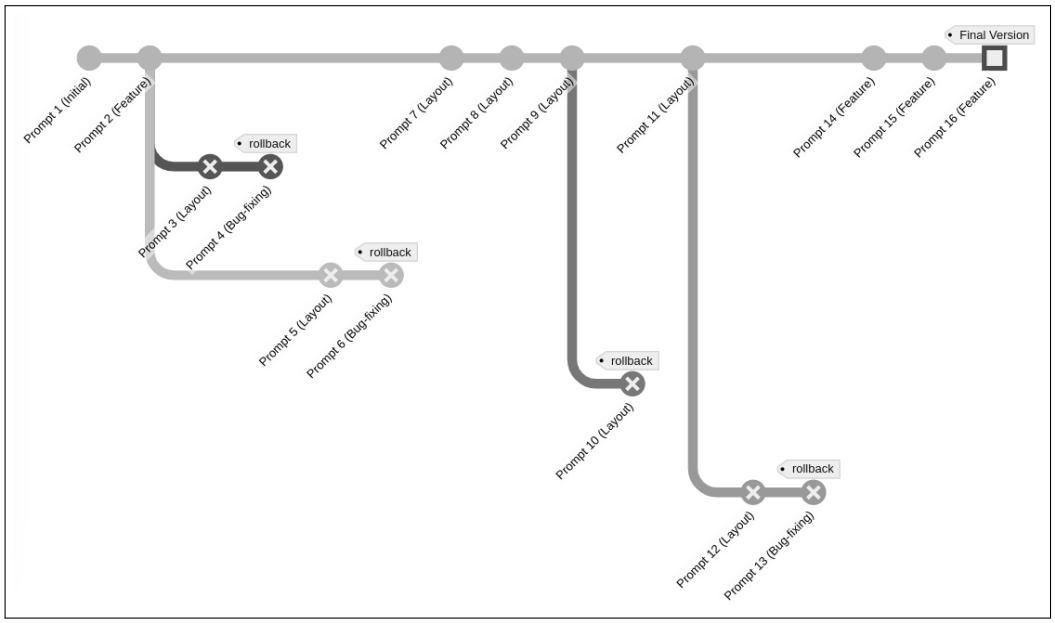

Figure 5.3: Rollbacks performed by Participant P2 in the pilot experiment (TodoApp).

Figures 5.4 and 5.5 show screenshots of the application created by one of the participants who completed all five stories. Figure 5.4 shows the question registration page, and Figure 5.5 shows the answer registration page.

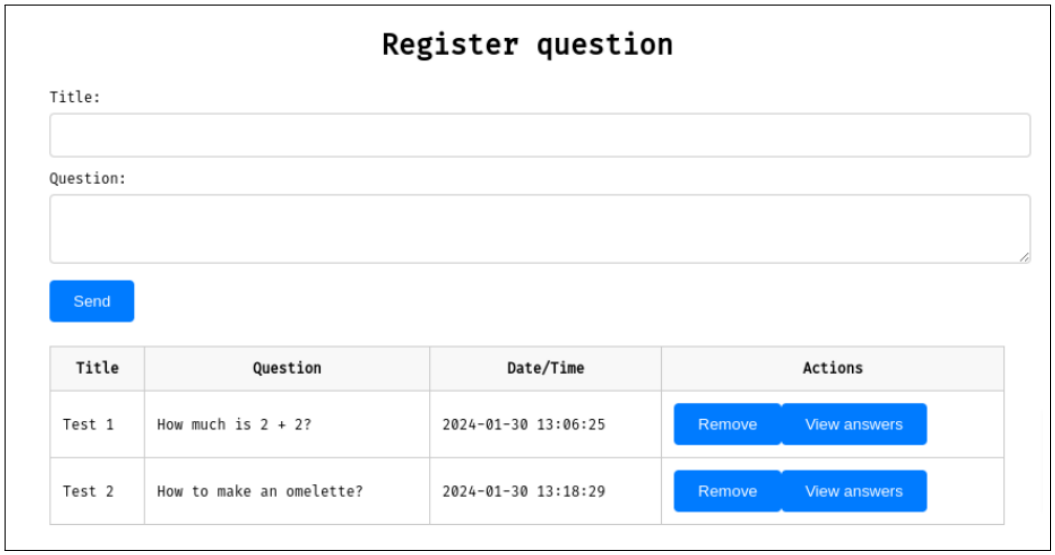

Figure 5.4: Question registration page created by participant P7 in the second experiment.

Table 5.4 shows the stories implemented by the participants in the second experiment. As we can see, participants P8, P9, and P10 were unable to implement the stories requesting answering questions and deleting answers. Participant P9 was the only one who failed to implement the third story, which defines that from the question page it should be possible to access the answers pages.

Figure 5.6 shows the number of prompts used by participants to build the ForumApp. Feature-type prompts were the most frequently used by participants, totaling

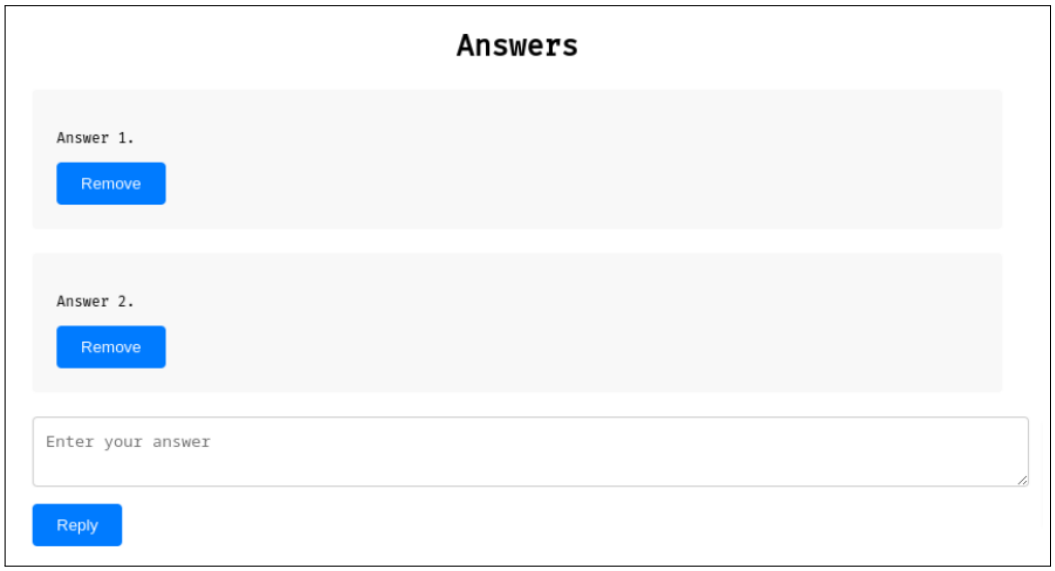

Figure 5.5: Answer registration page created by participant P7 in the second experiment.

User Stories	Participants who succeeded
As a user, I want to add a new question	All participants
As a user, I want to remove an existing question	All participants
As a user, I want to access the answers page	All participants, except P9
As a user, I want to answer a question	All participants, except P8, P9, P10
As a user, I want to delete an answer	All participants, except P8, P9, P10

Table 5.4: Participants who implemented each user story in the second experiment.

37 prompts, followed by 20 initial-type prompts. Each participant used two initial-type prompts: one for the page for registering questions and another for the page for registering answers. The bug-fixing and layout categories have 14 prompts each.

Of the seven participants who successfully implemented the application, P7 used the most prompts (12), while P3 used the fewest (5). The main reason for this difference seems to be the level of detail in their initial prompts. P3 started with a more detailed prompt, as presented next:

Prompt: *Create a page where I can register questions, where the registration includes a title, the text of the question, the logged-in user who asked the question, and the time the user asked. The questions should be listed in a table showing the title, the question text, the user who asked, and the date/time, alongside a button to delete the question and a button to view answers. The view answers button should go to an answer registration page for the selected question.*

On the other hand P7's initial prompt was more succint, as follows:

Participants P9 and P10 were the only ones who did not use layout-type prompts. This behavior is common among participants who struggle to finalize the application.

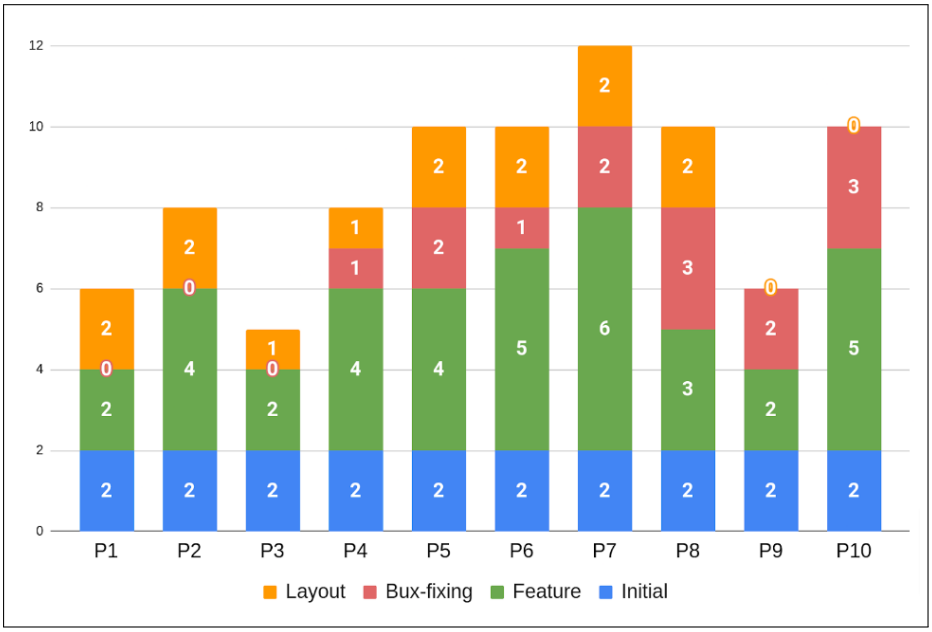

Figure 5.6: Total prompts used by each participant stacked by type.

Prompt: *Create a question registration page, which will have a box to enter the title of the question and add the description of the question. The questions will be listed separately in a table that will show the title of the question, the user who registered it, and the date and time.*

Since both participants were unable to implement all functionalities, they did not reach the stage where layout-type prompts are typically used, i.e., after everything is working as expected. Participants P1, P2, and P3 were the only ones who did not use bug-fixing prompts, likely because their more detailed initial prompts led to more precise results by the GPT model.

Finally, table 5.5 shows the number of rollbacks by each participant. As we can see, this feature was used less during the development of ForumApp (compared to the pilot study with TodoApp). Our hypothesis is that even though ForumApp has two pages, they are simpler than the single page specified in TodoApp. These simpler pages make it easier for the participants to define more precise and effective prompts. As a result, GPT was able to generate correct code in the first attempts and the need for rollbacks decreased.

Participant	**P1**	**P2**	**P3**	**P4**	**P5**	**P6**	**P7**	**P8**	**P9**	**P10**
Rollbacks	0	0	0	0	0	1	2	0	0	2

Table 5.5: Rollbacks by participants in the second experiment.

5.5 Participants' Perceptions

After the experiment, the participants of the first and second experiment (in total, 14 participants) also reported their perceptions about NoCodeGPT in a simple form. They answered two questions:

What are the tool's most positive points? Among the positive points highlighted by the participants were the simplicity of NoCodeGPT's interface (10 participants), the convenience of not having to copy-and-paste code (9 participants), and the lack of need for prior knowledge of programming languages (10 participants). Additionally, the ability to revert to previous versions and quickly view the prototype was seen as a valuable feature for those developing an application. Below are some of the positive aspects of the tool mentioned by the participants:

We were able to create a functional application without using any code, just simple instructions. The possibility of selecting previous versions was fundamental for correcting some bugs.

NoCodeGPT has an interesting purpose and is very useful, especially if you have a minimum knowledge of development. One of the best features is being able to go back to previous versions.

The tool is very easy to use; and the feature of not having to copy-and-paste any code or have prior knowledge of programming languages (although knowledge certainly increases is benefit); The feature to go back a version is very useful and helps a lot in the construction.

What are the downsides of the tool? The main negative point pointed out by participants was the delay in responses during interactions with the prompts (14 participants), as commented by the following participants:

NoCodeGPT at times takes too long to respond. Improve the response time.

Slow responses, I suggest increasing the processing speed of the tool.

It takes too long to execute the commands.

However, it is important to note that this delay refers to accessing the GPT API provided by OpenAI. Thus, it is a variable over which we have no control.

5.6 Lessons Learned

In the exploratory study, we concluded that the default interface offered by ChatGPT is not suitable for users without programming experience. Therefore, we decided to invest in the design and implementation of NoCodeGPT, a wrapper for the GPT API with features that make it easier for non-expert users to create a simple web application without writing code.

In this section, we reported the results of two studies designed to evaluate the effectiveness of NoCodeGPT in accomplishing its purpose. In the first study (TodoApp), two participants implemented all four proposed user stories, while the other two participants missed just one story. In the second study (ForumApp), seven participants implemented all five proposed user stories, three participants missed two stories, and one participant missed three stories.

Lesson Learned #6 : The results obtained with both apps showed that NoCodeGPT is effective in helping inexperienced users implement simple web applications without having to write code.

- In the exploratory study, using the default interface provided by ChatGPT, an inexperienced participant was unable to implement any of the proposed user stories.
- In the experiments described in this section, using the customized interface provided by NoCodeGPT, a scenario of total failure did not occur with any of the participants. Considering both experiments, NoCodeGPT was used by 14 inexperienced participants. More than half of such participants (9 participants) successfully completed the proposed applications, while the others completed at least half of the proposed user stories.

Next, we briefly discuss the role and contribution that the main features of NoCodeGPT had in these results.

- NoCodeGPT establishes a clear distinction between two categories of prompts: (1) prompts that define technologies and architectures; and (2) prompts that define functional requirements. The latter prompts are handled and encapsulated by the proposed tool. Consequently, users are solely responsible for writing prompts describing functional requirements.
- NoCodeGPT also manages and stores the code generated by GPT. This way, users do not need to copy and paste the generated code to a local directory or create a

specific folder structure for each application and architecture. These concerns are completely automated by the proposed tool. It also automates related tasks, such as installing libraries and configuring environment variables. It is worth mentioning that such tasks represented major obstacles for the non-expert participant from our Exploratory Study (Chapter 3).

- The rollback feature has been of major importance in NoCodeGPT's performance, particularly in the experiment with the TodoApp. In essence, this feature reduces the chances that users give up when a bug recurrently appears in the generated code. When this happens, users can readily restore a version that not have this bug.
- The code execution and visualization button was also very important for users to quickly check and identify issues in the generated code, including both bugs and layout problems.

5.7 Final Remarks

In this chapter, we conducting two comprehensive experiments with NoCodeGPT Tool. These experiments were carefully structured to assess the tool's effectiveness and involved a detailed process to ensure reliable and valid results. The outcomes of these experiments were promising, demonstrating a significant increase in the success rate of software development tasks when using NoCodeGPT.

The results highlighted not only the tool's ability to simplify the software construction process but also its potential to boost productivity. Users reported that the functionalities of NoCodeGPT were instrumental in streamlining their workflow, reducing the complexity typically associated with software development.

Furthermore, the experiments provided valuable insights into the tool's performance and user interaction, leading to several lessons learned.

The data related to the evaluation of the NoCodeGPT tool is at the link: `https://github.com/mauricioms/nocodegpt-evaluation`

Chapter 6

Conclusion

6.1 Contributions of the Dissertation

The increasing adoption of language models by software engineers emphasizes their transformative potential in the development of web applications. This dissertation investigated the application of ChatGPT as a tool for individuals with limited programming experience to create small web applications. The inadequacy of the standard GPT interface to effectively support this task was evidenced, due to its generic nature and lack of specific tools that facilitate the development process for non-technical users.

In response to this limitation, we developed a tool called NoCodeGPT, which consists of software specifically designed to improve the ease of development, allowing users lacking coding experience to build web applications effectively. The NoCodeGPT interface is designed to encapsulate and simplify all interaction with the language model, reducing users' need to elaborate complex prompts related to specific technologies or to manually manage complex aspects of the file structure.

The effectiveness of NoCodeGPT was evaluated through a rigorous experiment with 14 undegraduate students, whose results were quite promising. The experiment demonstrated that NoCodeGPT significantly outperformed ChatGPT's standard interface in terms of facilitation and effectiveness in building small web applications, without requiring the user's coding skills. For example, more than 60% of the participants successfully implemented a web system using only the functional requirements provided.

The results demonstrate the potential of NoCodeGPT as a powerful tool to democratize software development, allowing people without in-depth knowledge of programming to know and explore the world of web application creation. In addition, the creation of customized interfaces such as NoCodeGPT shows how the adaptation of language models can expand the accessibility of artificial intelligence technologies in various domains.

6.2 Future Work

For future work, there is a promising path for the continuous advancement and validation of the technologies discussed. One of the main focuses will be to broaden the scope of the evaluation, incorporating a more diverse range of applications and involving a larger number of developers in various development scenarios.

A crucial aspect of this development is the extension of NoCodeGPT compatibility to support a wider variety of programming languages and software architectures. This expansion will increase the versatility of the tool, allowing a wider audience to benefit from NoCodeGPT's capabilities. Interoperability with different systems and platforms will be an essential step to ensure that the technology can be integrated effectively into existing development environments. In addition, another area of exploration is the adaptation of NoCodeGPT for the development of different types of applications, with a special focus on mobile applications. Given the growing demand for mobile solutions, the ability to generate code for these platforms can open up new opportunities for developers, making it easier to create such applications.

Other possible future work that can be addressed is the validation of the creation of applications integrated into other systems and web platforms, allowing the tool to be used in more complex environments. This kind of tool evaluation will allow us to find out if it adapts to the non-functional requirements of software development properly.

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. GPT-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[2] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. A transformer-based approach for source code summarization. In Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel Tetreault, editors, *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 4998–5007, Online, July 2020. Association for Computational Linguistics.

[3] Nadia Alshahwan, Jubin Chheda, Anastasia Finegenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. Automated unit test improvement using large language models at meta. *CoRR*, abs/2402.09171, 2024.

[4] Owura Asare, Meiyappan Nagappan, and N. Asokan. Is GitHub's Copilot as bad as humans at introducing vulnerabilities in code? *Empiririical Software Engineering*, 28(6):129, 2023.

[5] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[6] Daniel Busch, Alexander Bainczyk, and Bernhard Steffen. Towards LLM-based system migration in language-driven engineering. In *International Conference on Engineering of Computer-Based Systems*, pages 191–200. Springer, 2023.

[7] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

[9] Min Du and Feifei Li. Spell: Streaming parsing of system event logs. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pages 859–864. IEEE, 2016.

[10] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[11] Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Ge Li, Zhi Jin, Xiaoguang Mao, and Xiangke Liao. Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning, 2024.

[12] Philip Guo. Real-real-world programming with ChatGPT, 2023. Available online at: `https://www.oreilly.com/radar/real-real-world-programming-with-chatgpt/`, last accessed on Oct 2023.

[13] Nathan Hagel, Nicolas Hili, and Didier Schwab. Turning Low-Code development platforms into true No-Code with LLMs. In *MODELS Companion'24*, 2024.

[14] Matthias Harter. LLM assisted no-code hmi development for safety-critical systems.

[15] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R. Lyu. Drain: An online log parsing approach with fixed depth tree. In *2017 IEEE International Conference on Web Services (ICWS)*, pages 33–40, June 2017.

[16] Amr Hendy, Mohamed Abdelrehim, Amr Sharaf, Vikas Raunak, Mohamed Gabr, Hitokazu Matsushita, Young Jin Kim, Mohamed Afify, and Hany Hassan Awadalla. How good are GPT models at machine translation? a comprehensive evaluation. *arXiv preprint arXiv:2302.09210*, 2023.

[17] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*, 2023.

[18] Zhen Ming Jiang, Ahmed E Hassan, Parminder Flora, and Gilbert Hamann. Abstracting execution logs to execution events for enterprise applications (short paper).

In *2008 The Eighth International Conference on Quality Software*, pages 181–186. IEEE, 2008.

[19] Kailun Jin, Chung-Yu Wang, Hung Viet Pham, and Hadi Hemmati. Can ChatGPT Support Developers? An Empirical Evaluation of Large Language Models for Code Generation. In Diomidis Spinellis, Alberto Bacchelli, and Eleni Constantinou, editors, *21st IEEE/ACM International Conference on Mining Software Repositories, MSR 2024, Lisbon, Portugal, April 15-16, 2024*, pages 167–171. ACM, 2024.

[20] Caitlin Kelleher and Randy Pausch. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM computing surveys (CSUR)*, 37(2):83–137, 2005.

[21] Edwin C.Y. Koh. Auto-dsm: Using a large language model to generate a design structure matrix. *Natural Language Processing Journal*, 9:100103, 2024.

[22] Van-Hoang Le and Hongyu Zhang. An evaluation of log parsing with ChatGPT. *CoRR*, abs/2306.01590, 2023.

[23] Van-Hoang Le and Hongyu Zhang. Log parsing with prompt-based few-shot learning. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, page 2438–2449. IEEE Press, 2023.

[24] Yue Liu, Thanh Le-Cong, Ratnadira Widyasari, Chakkrit Tantithamthavorn, Li Li, Xuan-Bach Dinh Le, and David Lo. Refining ChatGPT-generated code: Characterizing and mitigating code quality issues. *CoRR*, abs/2307.12596, 2023.

[25] Antonio Mastropaolo, Luca Pascarella, Emanuela Guglielmi, Matteo Ciniselli, Simone Scalabrino, Rocco Oliveto, and Gabriele Bavota. On the robustness of code generation techniques: An empirical study on GitHub Copilot. In *45th IEEE/ACM International Conference on Software Engineering*, pages 2149–2160, 2023.

[26] Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais, and Zhen Ming (Jack) Jiang. GitHub Copilot AI pair programmer: Asset or liability? *Journal of Systems and Software*, 203:111734, 2023.

[27] A.C. Müller and S. Guido. *Introduction to Machine Learning with Python: A Guide for Data Scientists*. O'Reilly Media, 2016.

[28] Vijayaraghavan Murali, Chandra Shekhar Maddila, Imad Ahmad, Michael Bolin, Daniel Cheng, Negar Ghorbani, Renuka Fernandez, and Nachiappan Nagappan. CodeCompose: A large-scale industrial deployment of AI-assisted code authoring. *CoRR*, abs/2305.12050, 2023.

[29] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. Using an LLM to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[30] Nhan Nguyen and Sarah Nadi. An empirical evaluation of GitHub Copilot's code suggestions. In *19th International Conference on Mining Software Repositories (MSR)*, pages 1–5, 2022.

[31] Anh Nguyen-Duc, Beatriz Cabrero-Daniel, Adam Przybylek, Chetan Arora, Dron Khanna, Tomas Herda, Usman Rafiq, Jorge Melegati, Eduardo Guerra, Kai-Kristian Kemell, Mika Saari, Zheying Zhang, Huy Le, Tho Quan, and Pekka Abrahamsson. Generative artificial intelligence for software engineering – a research agenda, 2023.

[32] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback, 2022.

[33] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan -Gavitt, and Ramesh Karri. Asleep at the keyboard? assessing the security of GitHub Copilot's code contributions. In *43rd IEEE Symposium on Security and Privacy*, pages 754–768, 2022.

[34] Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. The impact of AI on developer productivity: Evidence from GitHub Copilot. *CoRR*, abs/2302.06590, 2023.

[35] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 1(8):9, 2019.

[36] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Pearson, 2016.

[37] Ahmed R. Sadik, Antonello Ceravola, Frank Joublin, and Jibesh Patra. Analysis of ChatGPT on source code. *CoRR*, abs/2306.00597, 2023.

[38] Jiho Shin, Clark Tang, Tahmineh Mohati, Maleknaz Nayebi, Song Wang, and Hadi Hemmati. Prompt engineering or fine tuning: An empirical assessment of large language models in automated software engineering tasks, 2023.

[39] Mohammed Latif Siddiq, Joanna C. S. Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinicius Carvalho Lopes. Exploring the effectiveness of large language models in generating unit tests. *CoRR*, abs/2305.00418, 2023.

[40] Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. An analysis of the automatic bug fixing performance of ChatGPT. In *IEEE/ACM International Workshop on Automated Program Repair (APR@ICSE 2023)*, pages 23–30. IEEE, 2023.

[41] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. Intellicode compose: Code generation using transformer, 2020.

[42] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. An empirical study on learning bug-fixing patches in the wild via neural machine translation, 2019.

[43] Xuheng Wang, Xu Zhang, Liqun Li, Shilin He, Hongyu Zhang, Yudong Liu, Lingling Zheng, Yu Kang, Qingwei Lin, Yingnong Dang, et al. Spine: a scalable log parser with feedback guidance. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1198–1208, 2022.

[44] Jules White, Sam Hays, Quchen Fu, Jesse Spencer-Smith, and Douglas C. Schmidt. ChatGPT prompt patterns for improving code quality, refactoring, requirements elicitation, and software design, 2023.

[45] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. Automated program repair in the era of large pre-trained language models. In *45th IEEE/ACM International Conference on Software Engineering*, pages 1482–1494, 2023.

[46] Hu Xu, Bing Liu, Lei Shu, and Philip S Yu. Dombert: Domain-oriented language model for aspect-based sentiment analysis. *arXiv preprint arXiv:2004.13816*, 2020.

[47] Ting Zhang, Bowen Xu, Ferdian Thung, Stefanus Agus Haryono, David Lo, and Lingxiao Jiang. Sentiment analysis for software engineering: How far can pre-trained transformer models go? In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 70–80. IEEE, 2020.