



Using AI as a Development Accelerator

RUI PEDRO MESQUITA SOARES

Junho de 2024

Using AI as a Development Accelerator

Rui Pedro Mesquita Soares

**Dissertation for the Master's Degree in
Computer Engineering, Specialization in
Software Engineering**

Advisor: Fernando Jorge Ferreira Duarte

Supervisor: Roberto Pimentel Silva

Jury:

President:

Luís Miguel Moreira Lino Ferreira, Professor Coordenador, DEI/ISEP

Members:

António Constantino Lopes Martins, Professor Adjunto, DEI/ISEP

Declaration of Integrity

I declare that I have conducted this academic work with integrity.

I have not plagiarized or applied any form of misuse of information or falsification of results throughout the process that led to its elaboration.

Therefore, the work presented in this document is original and under my authorship. It has not been used previously for any other purpose.

I also declare that I fully know P. PORTO's Code of Ethical Conduct.

ISEP, Porto, 30 de junho de 2024

Resumo

Esta dissertação investiga o potencial transformador da Inteligência Artificial (IA) na aceleração e automação do processo de desenvolvimento de microserviços em Java, com um foco específico no framework Spring Boot. À medida que o desenvolvimento de software se torna cada vez mais complexo, a integração de tecnologias de IA promete simplificar tarefas de codificação, assegurando que o código gerado atende tanto aos padrões técnicos quanto aos requisitos de negócio.

O estudo inicia-se com uma revisão exaustiva da literatura existente sobre geração de código impulsionada por IA, explorando os avanços recentes e identificando as falhas dos conhecimentos atuais. Esta revisão fornece a base teórica necessária para a fase seguinte do trabalho, que envolve a experimentação prática com várias ferramentas de IA, como o GitHub Copilot, TabNine, e diferentes versões do ChatGPT. Estas ferramentas são testadas em cenários de desenvolvimento real, avaliando a sua eficácia na geração de código funcional para microserviços em Java, a sua integração com APIs abertas e a sua capacidade de manter padrões de qualidade.

O principal objetivo da investigação é desenvolver uma metodologia baseada em IA para gerar código específico para um microserviço Java dentro do ambiente Spring Boot. Esta metodologia inclui a criação de APIs abertas, a integração com uma base de dados e a ligação a serviços REST externos. Para alcançar este objetivo, foram definidos objetivos específicos como:

- **Revisão da Literatura:** Realizar uma revisão abrangente da literatura sobre geração de código com IA, identificando as técnicas mais eficazes e lacunas de conhecimento.
- **Comparação e Seleção de Ferramentas de IA:** Conduzir uma análise comparativa detalhada de várias ferramentas de IA disponíveis para geração de código, avaliando suas vantagens e limitações no contexto da investigação.
- **Avaliação de Desempenho:** Avaliar a eficácia das metodologias de IA selecionadas em termos de precisão, eficiência e utilidade no processo de geração de código, utilizando métricas objetivas e cenários de teste práticos.
- **Desenvolvimento de um Estudo de Caso Funcional:** Definir e implementar um estudo de caso funcional de complexidade média a alta para o microserviço alvo.
- **Análise Comparativa com Métodos Tradicionais:** Comparar a abordagem impulsionada por IA com métodos tradicionais de geração de código, destacando as forças e fraquezas da IA, neste contexto.
- **Autonomia da IA em Diferentes Etapas:** Determinar o nível de autonomia da ferramenta de IA em várias etapas do desenvolvimento, incluindo a criação de histórias de utilizador, a geração de código correspondente, e a integração de componentes de observabilidade e processos CI/CD.
- **Análise dos Resultados:** Analisar os resultados da investigação para extrair conclusões significativas sobre a eficácia e praticidade da IA na geração de código.

- Propostas de Melhorias e Futuras Pesquisas: Propor melhorias à metodologia de geração de código baseada em IA e identificar áreas potenciais para futuras investigações.
- Identificação de Questões de Segurança e Privacidade: Reconhecer e resolver vulnerabilidades de segurança e questões de privacidade no processo de geração de código impulsionado por IA.

Além disso, a tese aborda as implicações mais amplas do uso de IA no desenvolvimento de software, incluindo preocupações com a privacidade dos dados, segurança e impacto no emprego no sector da engenharia de software. A investigação considera também as questões éticas envolvidas, como a necessidade de garantir que o código gerado pela IA é justo e equitativo, evitando perpetuar ou amplificar preconceitos existentes.

A metodologia adotada é reflexiva e iterativa, ajustando-se às descobertas e desafios emergentes ao longo do processo. Sessões regulares de discussão e feedback com a equipa do projeto, permitiram refinar a aplicação e realizar a avaliação da ferramenta de IA, assegurando que a investigação se mantém centrada nas necessidades práticas e reais do mundo do desenvolvimento de software.

Em última análise, esta pesquisa visa proporcionar uma abordagem equilibrada para a integração da IA no desenvolvimento de software, destacando os seus benefícios e identificando áreas para futuras pesquisas. As conclusões esperam contribuir para uma maior compreensão de como as tecnologias de IA podem revolucionar as práticas de desenvolvimento de software, melhorando a eficiência, reduzindo o tempo de desenvolvimento e garantindo a qualidade do código gerado.

Abstract

This dissertation explores the transformative potential of Artificial Intelligence (AI) in accelerating and automating the development of Java microservices, focusing on the Spring Boot framework. As software development becomes increasingly complex, AI integration promises to simplify coding tasks while meeting technical standards and business requirements.

The study begins with a comprehensive literature review on AI-driven code generation, identifying recent advancements and current knowledge gaps. Practical experimentation follows, involving tools like GitHub Copilot, TabNine, and various versions of ChatGPT to assess their effectiveness in generating functional Java microservices code, integrating open APIs, and maintaining quality standards.

The primary goal is to develop an AI-based methodology for generating specific code for a Java microservice within the Spring Boot environment. This goal involves creating open APIs, integrating databases, and connecting to external REST services.

The thesis also examines the broader implications of AI in software development, such as data privacy, security, and employment impacts. It considers ethical issues to ensure fair and unbiased code generation. The adopted methodology is reflective and iterative, adjusting to emerging findings and challenges, with regular project team discussions refining AI tool application and evaluation.

In conclusion, this research takes a comprehensive approach to AI integration in software development, highlighting its benefits and identifying areas for future research. The findings are expected to significantly contribute to the understanding of how AI technologies can revolutionize software development practices, enhancing efficiency, reducing development time, and ensuring code quality.

Keywords: Artificial Intelligence, AI, Machine Learning, Software Development, Code Generation, Automation, Java, Microservices, Spring Boot, AI Tools

Acknowledgments

I want to thank my parents, who, throughout my life, have given me all the values I need to achieve all my goals. They have always believed in me and my performance. They have always provided support and the necessary conditions for me to complete this stage with skill, comfort, and joy.

I want to thank my grandparents, who were one of my sources of motivation right from the start. They showed me that with will and perseverance, anything is possible, and they were always there when I needed them most.

Beatriz Duarte, my girlfriend and best friend, has been with me for seven years. She has been there for me throughout my academic career and has always given me all the support, affection, joy, and inspiration I needed. Thank you for all the hours of fun, the patience she had with me, the moments we spent together, and for helping me become a better person. This has strengthened our friendship and relationship, enabling us to achieve everything together.

To Diogo Rodrigues, my best friend, who has been with me for 13 years, has been there for half my life, has helped me through good times and bad, but for the support and well-being he has always been willing to share.

To the teachers at ISEP, especially Professor Fernando Jorge Ferreira Duarte, my supervisor, who, as well as showing a genuine enthusiasm for teaching, managed to reinforce my interest in this area through the way he lectured and his determination to solve any problems that arose, allowing me to develop new technical skills. His ability to impart knowledge ignited a passion for learning and equipped me with the tools to tackle complex challenges. The professor had already been my supervisor during the preparation of the internship project document at the end of my degree, providing an enriching experience that cemented my professional and academic skills. With deep gratitude, I thank him again for his tireless support and insightful guidance, which proved crucial during the internship stage and in the final phase of my master's degree. His constant dedication and enthusiasm have inspired me to pursue excellence in my academic and professional career.

Finally, to Celfocus and the teams that welcomed me, to Roberto Pimentel Silva, my supervisor, who advised and guided me in the development of the project and always showed friendship and professionalism in the countless meetings, to Pedro Carrilho Martins, my Team Leader, who integrated me into the team in the best possible way, who helped me improve both personally and professionally, and finally to the various colleagues in the team who were always available if any doubts or problems arose and made me feel "at home."

Index

1	Introduction	1
1.1	Background and Context	1
1.2	Description of the Problem	2
1.2.1	Objectives	3
1.2.2	Approach	4
1.2.3	Contributions	4
1.2.4	Work Planning	5
1.3	Significance of the Study	6
1.4	Thesis structure	6
2	Literature Review	9
2.1	State of the Art	9
2.1.1	Evaluating AI's Role in Programming Education	10
2.1.2	Analyzing the Quality and Robustness of AI-Generated Code	12
2.1.3	Exploring Intellectual Property Concerns in AI-Generated Code	13
2.1.4	Investigating Security Aspects in AI-Enabled Code Generation	14
2.1.5	Assessing the Implications for Software Engineering Practices	16
2.2	Ethical Considerations	17
2.2.1	Celfocus Ethics Regarding Generative AI	18
3	Methodology	21
3.1	Research Process	21
3.1.1	Research Questions	21
3.1.2	Data Sources	22
3.1.3	Search Terms	22
3.1.4	Eligibility Criteria	23
3.1.5	Data Collection Process	23
3.1.6	Results	24
3.2	Tool Comparison Criteria	25
4	Case Studies and Experimentation	27
4.1	Tools Used	27
4.2	Behavioral Patterns and Predictions in Companies	28
4.3	Polyvalence vs Monovalence in Tools	30
4.4	Prompt Engineering Analysis	31
5	Results	34
5.1	Data Collection Procedure	34
5.1.1	Data Sources	34
5.1.2	Evaluating AI Tools for Code Generation	35

5.1.3	Data Collection Challenges	36
5.1.4	Data Collection Summary	36
5.2	Data Interpretation	37
5.2.1	Performance Efficiency	37
5.2.2	User Experience	45
5.2.3	Integration Capability	49
5.2.4	Cost-effectiveness	52
5.2.5	Scalability	55
5.2.6	Security and Reliability.....	58
5.2.7	Summary of Findings.....	61
5.3	Prompt Effectiveness and Construction.....	61
5.3.1	Importance of Prompt Engineering	61
5.3.2	Analysis of Prompt Effectiveness	62
5.3.3	Impact of Prompt Design on Results.....	63
5.3.4	Best Practices for Prompt Engineering.....	63
5.3.5	Summary of Findings.....	64
6	Discussion	66
6.1	Integration of AI in Programming	66
6.1.1	Enhanced Productivity and Efficiency.....	66
6.1.2	Collaborative Coding.....	67
6.1.3	Evolving Role of Developers	67
6.1.4	Challenges and Opportunities	67
6.1.5	Summary.....	68
6.2	Objectives Fulfillment.....	71
6.3	Limitations of the Study	68
6.3.1	Scope of Tools Evaluated	68
6.3.2	Sample Size and Diversity.....	68
6.3.3	Focus on Java Microservices	69
6.3.4	Rapid Evolution of AI Technologies	69
6.3.5	Context-Specific Findings	69
7	Conclusion	71
7.1	Summary of Findings.....	72
7.2	Practical Implications	74
7.2.1	For Businesses	74
7.2.2	For Engineers	74
7.2.3	Summary.....	75
8	Future Work	76
8.1	Recommendations for Future Research.....	76
8.1.1	Expanding the Range of Evaluated Tools	76
8.1.2	Redo the Survey	76
8.1.3	Enhancing Sample Diversity and Size.....	77
8.1.4	In-Depth Analysis of Long-Term Impacts.....	77
8.1.5	Advanced Security and Ethical Research	77

8.1.6	Optimizing Workflow Integration	77
8.1.7	Educational Initiatives and Impact.....	77
8.1.8	Exploring AI-Assisted Collaborative Development	78
8.1.9	Summary.....	78
8.2	Future Trends to Watch	78
8.2.1	Advanced Natural Language Processing.....	78
8.2.2	AI-Driven Code Refactoring and Optimization	78
8.2.3	Integration of AI in DevOps	78
8.2.4	AI-Augmented Debugging and Testing.....	79
8.2.5	Summary.....	79

List of Figures

Figure 1 Flowchart of the Data Collection Process	24
Figure 2 Times for Test Implementing	38
Figure 3 Times for Integrating Database Connections.....	39
Figure 4 Times for Developing APIs.....	39
Figure 5 Times for Altering Swagger Documentation	40
Figure 6 Times for Configuring Jenkins Pipelines.....	40
Figure 7 Times for Improving Existing Code.....	41
Figure 8 Number of Errors and Issues When Implementing Tests	41
Figure 9 Number of Errors and Issues When Integrating DB Connections.....	42
Figure 10 Number of Errors and Issues When Developing APIs	42
Figure 11 Number of Errors and Issues When Altering Swagger Documentation.....	43
Figure 12 Number of Errors and Issues When Configuring Jenkins Pipelines.....	43
Figure 13 Number of Errors and Issues When Improving Existing Code	44
Figure 14 Familiarity with AI-Powered Code Assistance Tools	45
Figure 15 AI-Powered Code Assistance Tool Usage	45
Figure 16 Coding Productivity Answers	46
Figure 17 Code Quality Answers	47
Figure 18 Security and Privacy Answers	47
Figure 19 Learning Curve Answers.....	48
Figure 20 Adoption Willingness Answers.....	48
Figure 21 Celfocus Profits Without Indirect Costs	54
Figure 22 Lack of Context Result.....	62
Figure 23 “God Prompt” for Prompt Generation.....	64

List of Tables

Table 1 - Data sources	22
Table 2 Compatibility with IDEs and Operating Systems Resume .Error! Bookmark not defined.	
Table 3 API Connectivity and Plugin Support Resume	50
Table 4 Embedding into CI/CD Pipelines Resume	51
Table 5 Tools Direct Cost.....	52
Table 6 Indirect Costs	53
Table 7 Cost-Benefit Analysis	53
Table 8 Handling Large-Scale Projects Resume	56
Table 9 System Resource Utilization Resume	57
Table 10 Stress Tests Resume	59
Table 11 Historical Security Issues Resume	60

List of Abbreviations

List of Acronyms

AI	Artificial Intelligence
CAD	Computer-Aided Design
CGBE	Code Generation By Example
CI/CD	Continuous Integration/Continuous Deployment
CWE	Common Weakness Enumeration
GDPR	General Data Protection Regulation
GenAI	Generative Artificial Intelligence
IA	Inteligência Artificial
IEEE	Institute of Electrical and Electronics Engineers
ISO/IEC	International Organization for Standardization/International Electrotechnical Commission
LLCG	Large Language Model-based Code Generation
MDE	Model-Driven Engineering
PCGM	Pre-trained Code Generation Models
RGPD	Règlement Général sur la Protection des Données (General Data Protection Regulation in English)

1 Introduction

This thesis follows in the context of the DIMEI - Dissertation course unit and aims to describe and contextualize the solution proposed by the Celfocus team and the entire process involved in its development.

1.1 Background and Context

In recent years, software development has experienced profound changes driven by technological patterns' evolution and market demands' active nature. A pivotal element in this transformative scene is the arrival of artificial intelligence (AI) technologies, signaled as catalysts for expediting development processes. The requirement for AI's incorporation has been amplified considering the escalating complexity and scalability requisites characteristic of contemporary software systems. Such intricacies have begun to outdo the capabilities of conventional development methodologies, thereby necessitating the adoption of more sophisticated and efficient approaches to code generation.

AI-based methodologies are posited as a viable response to the escalating quest for expedited software development processes without compromising quality. Nevertheless, integrating AI in software development introduces distinct challenges, particularly in generating functional code. A pivotal concern is the capacity of AI-generated outputs to conform to the rigorous functional and technical specifications necessitated by contemporary business ecosystems. This apprehension finds resonance in recent scholarly investigations, such as the study by (Losbichler and Lehner, 2021), which underscores the existing limitations of AI systems in interpreting and applying complex business logic.

Furthermore, the level of human involvement necessary to refine AI-generated code, ensuring it adheres to specific requirements, is a topic of ongoing academic investigation. This ambiguity underscores the limitations inherent in present-day AI tools. Despite their advanced nature, these tools still necessitate considerable human supervision, as elaborated in the work (Yetiştiren et al., 2023a). The challenges associated with this are not merely technical. They also include the need for conformity with established industry standards, ensuring similarity with user and business requirements, and effectively integrating these AI solutions into pre-existing software development workflows.

Financial and resource constraints significantly influence this landscape. As exposed in (Reilly, 2023) the economic weight associated with incorporating and training AI systems presents a substantial obstacle, predominantly for smaller organizations. Another complicating factor is the limited pool of AI and software engineering professionals. This reality may impede AI's rapid adoption and improvement in software development procedures.

The industry's identification and response to these challenges and limitations vary distinctly. Larger enterprises and technology giants, such as (Bhati, 2023) observed, are actively advancing in AI-driven software development initiatives. On the contrary, smaller firms and traditional businesses exhibit a more reserved approach. This cautious stance often stems from their internal resource constraints and a need for more specialized knowledge, which poses a barrier to embracing such technological advancements.

This thesis explores the significant impacts of using AI in code generation, emphasizing Java microservices development. It examines the utilization of AI in this area, all while recognizing the complexities and challenges established in the current landscape. This exploration aims to show the opportunities and constraints of AI applications in software development.

1.2 Description of the Problem

The core challenge addressed in this thesis revolves around the integration and effectiveness of AI-driven tools in generating functional code for Java microservices within the Spring Boot framework. Despite the rapid advancement of AI technologies, their application in code generation presents several significant obstacles. However, AI has immense potential to produce code that meets modern business applications' rigorous functional and technical requirements. While concerns regarding the sufficiency of AI-generated code quality and its adherence to precise specifications are dominant in the software development community, the future of AI in code generation is promising.

A critical issue is the need for substantial human intervention to refine AI-generated outputs to ensure they align with specific coding standards and project goals. This requirement for ongoing human oversight not only raises questions about the efficiency gains offered by AI tools and their practicality in everyday development tasks but also underscores the crucial role of software developers in the process. Integrating AI into established development workflows poses a complex challenge, demanding technical alignment and compatibility with organizations' operational and strategic goals.

Furthermore, the reliability of AI in accurately interpreting complex business logic without explicit and extensive training remains uncertain. This uncertainty is intensified by the economic and resource constraints many organizations face, particularly smaller firms, which may need more expertise and financial capacity to leverage AI effectively.

This thesis addresses these challenges by evaluating AI's performance and practicality in automating the code generation process, particularly for Java microservices. It seeks to identify which AI methodologies can be effectively integrated into existing systems and workflows, assess their real-world utility, and determine the extent to which they can reduce the dependency on manual coding and accelerate development timelines without compromising quality or increasing the risk of errors.

1.2.1 Objectives

The primary objective of this research is to develop and critically evaluate an AI-based methodology for generating code tailored explicitly for a Java microservice within the Spring Boot framework. This methodology will encompass the development of Open APIs alongside the integration of a database and external REST services. The following specific objectives have been established to achieve the goal:

1. **Conduct a Comprehensive Literature Review:** Conduct an exhaustive review of the existing literature concerning AI-driven code generation. The review will identify the most recent advancements, pinpoint the most effective techniques, and uncover gaps in the current body of knowledge.
2. **Comparison and Selection of AI Tools:** Conduct a detailed comparative analysis of various AI tools available for code generation. This comparison will focus on the advantages and limitations of each tool in the context of our specific research scenario, ending with selecting the most appropriate tool for our needs.
3. **Performance Evaluation of AI-based Approaches:** This evaluation will assess the effectiveness of the chosen AI methodologies in accuracy, efficiency, and utility in the code generation process. It will be grounded in objective metrics and substantiated through practical testing scenarios.
4. **Development of a Functional Case Study:** The targeted microservice will require the definition of a functional case study of medium-to-high complexity.
5. **Comparative Analysis with Existing Methods:** This section compares the AI-driven approach against traditional code generation methods, highlighting AI's strengths and weaknesses in this context.
6. **Assessing AI Autonomy in Various Stages:** Use of the selected AI tool to determine its level of autonomy in various stages of development, including:
 - Creation of user stories based on functional requirements.
 - Generate code that fulfills these user stories.
 - Integration of observability components, API documentation, and CI/CD processes.
7. **Analyzing Research Outcomes:** Analyze the research findings to extract meaningful insights and conclusions regarding the effectiveness and practicality of AI in code generation.
8. **Propose Improvements and Future Research Directions:** Based on the research results, propose enhancements to the AI code generation methodology and identify potential avenues for future exploration in this domain.
9. **Identification of Security and Privacy Concerns:** Recognition and resolution of any security vulnerabilities and issues in handling privileged customer information within the AI-driven code generation process.
10. **Comprehensive Documentation:** Compiling and writing a thorough and coherent research thesis that meticulously documents all stages of the research, from the literature review to the conclusions and recommendations.

These objectives are designed to ensure a holistic and rigorous examination of AI's capabilities and limitations in the context of Java microservice code generation while also considering ethical, security, and practical aspects.

1.2.2 Approach

The approach to achieving the proposed research objectives was structured into three distinct phases, which align with the practical project environment facilitated by team collaboration. However, it is important to note that the research was personal.

The first stage, a comprehensive literature review, was not just a preliminary step but a crucial one. It was instrumental in understanding the current state of AI-driven code generation, for Java microservices developed using the Spring Boot framework. The literature review aimed to identify and highlight recent advances, practical approaches, and existing knowledge gaps that would guide the thesis's ensuing stages, underscoring its importance in shaping the subsequent research.

The second phase focused on comparing and selecting AI tools for code generation tailored to Java microservices. This process entailed a detailed review and analysis of several AI tools, considering their advantages and disadvantages to determine which would be the most appropriate tool considering the needs presented by the research. The selected AI methodology was then poised on further project-setting tests and evaluations.

The last step involved applying the developed AI tool within a real-world project environment. This included evaluating the AI methodology's performance using objective metrics and practical testing scenarios to determine its accuracy, efficiency, and usability. With this project, it was possible to compare the AI-driven approach with traditional code generation methods and explore the level of autonomy the AI tool has in different development processes. These processes involved code generation that fulfilled user stories, observability components, API documentation, and CI/CD process integrations.

This research adopts a highly reflective and iterative approach, where the methodologies and objectives are changed depending on the findings and challenges emerging during the process. Regular discussion and feedback sessions with the project team refined the application and evaluation of the AI tool, ensuring that the research was human-centered in its motivation and guided by practical, real-world application. This approach ensures that the thesis comprehensively covers theoretical study and practical application insight in a collaborative yet individually driven manner.

1.2.3 Contributions

The completion of this thesis has led to a comprehensive evaluation of multiple AI-driven code generation tools, such as ChatGPT, GitHub Copilot, and Tabnine, tailored for Java microservices within the Spring Boot framework. This research significantly contributes to the

field by providing a detailed comparative analysis of these tools, highlighting their strengths, limitations, and suitability for specific programming environments. The study has identified the most effective AI methodologies, providing a critical resource for developers and researchers interested in AI-based code generation.

One of this thesis's key contributions is establishing objective criteria for evaluating AI tools in the context of code generation. These criteria, used to assess each tool's accuracy, efficiency, and utility, offer a systematic approach to choosing the best tool based on a project's needs. This methodology provides a solid foundation for decision-making, instilling confidence and reassurance in organizations and individual developers when integrating AI into their development pipelines.

This research goes beyond theoretical analysis to highlight the transformative potential of implementing AI tools in real-world scenarios. It analyzes their impact on development workflows, their ability to enhance productivity, and their potential to reduce manual coding requirements. The findings from this thesis provide valuable insights into how AI can revolutionize software development practices and introduce innovations in code generation.

1.2.4 Work Planning

The planning of this thesis was a collaborative effort, carefully structured to ensure a thorough evaluation of AI-driven code generation tools within the established timelines. This process involved a close partnership with the thesis supervisor to define the research's scope, methodologies, and milestones. To effectively manage the complexity of the tasks, the Gantt chart was divided into two distinct parts: Design and Implementation and Experimentation and Evaluation.

The initial phase of this thesis was dedicated to laying the groundwork for the research. These activities were crucial in setting up the AI tool selection and assessment criteria. The detailed Gantt chart for this phase, outlining the timeline for each activity, can be found in Annex 1 - Design and Implementation

of the thesis.

The second phase involved the practical application of the research design. With its focus on real-world application, this phase was critical in understanding each tool's practical implications and effectiveness in a real-world setting and exciting in terms of the potential impact of our research. The Gantt chart for this phase, detailing the timeline for experimentation and the evaluation process, is presented in Annex 2 - Experimentation and Evaluation

Annex 2 - Experimentation and Evaluation

.

Gantt charts were vital tools in managing the project timeline effectively. They visually represented each stage of the research process, providing a clear roadmap. They highlighted critical periods for data collection, analysis, evaluation, and final synthesis, ensuring that each aspect of the thesis was given adequate attention and completed on schedule.

1.3 Significance of the Study

The central issue explored in this research involves assessing the viability and effectiveness of AI technologies in generating code for Java microservices within the Spring Boot framework. The complexity of this challenge extends beyond the technical aspects of selecting and applying suitable AI models and techniques for code generation. It also involves extensive considerations, such as ensuring the compatibility of AI-generated code with established coding standards and practices.

A critical element of this assessment is evaluating AI's performance based on objective criteria, including accuracy, efficiency, and compliance with best practices. This evaluation encompasses the perspectives and interests of various stakeholders, such as developers, project managers, and end-users, to ascertain that AI integration aligns with their needs and expectations.

From an ethical standpoint, the study will confront issues of bias in AI-generated code to ensure fairness and equity. Adherence to severe ethical guidelines in data handling, focusing on maintaining privacy and confidentiality, is paramount in maintaining legal and social responsibilities.

The methodology employed in understanding the problem incorporates diverse perspectives, avoiding preconceived notions and biases. It involves examining scientifically validated evidence and credible data about the use of AI in software development. The analysis will present a comprehensive and precise view of AI's role and efficacy in the domain, ensuring the information gathered reflects the latest advancements and techniques.

The consequences of failing to integrate AI effectively into software development are significant. With these advancements, the industry can manage the growing complexity of software systems, which leads to inefficiencies and a loss of competitive edge. Conversely, successfully addressing this challenge could result in more efficient development processes, quicker time-to-market, and enhanced software quality.

In summary, the problem is approached from a multi-dimensional perspective, involving technical, ethical, and stakeholder considerations. This approach helps ensure that the potential ramifications of incorporating AI into the software development lifecycle are thoroughly comprehended and addressed.

1.4 Thesis structure

This thesis is organized into eight main chapters, each designed to systematically address the multifaceted aspects of AI-driven code generation for Java microservices within the Spring Boot framework. The structure facilitates a logical progression of the research and mirrors the comprehensive approach taken to explore, analyze, and evaluate the integration of AI in software development.

The first chapter provides an overview of the thesis, setting the stage for the following detailed examination. It introduces the context of the research, the significant challenges currently faced in the field, and the potential of AI to address these issues. It also outlines the research's objectives and briefly describes the subsequent chapters.

The second chapter divides the literature review into two sections. The first, 'State of the Art,' delves into current AI-driven code generation advancements and methodologies, identifying achievements and gaps. The second, 'Ethical Considerations,' discusses the ethical implications of using AI in software development, underscoring the importance of moral integrity in automated systems.

The third chapter describes the methodology employed in the research, detailing the processes used to gather, analyze, and interpret data. It includes sub-sections on research questions, data sources, search terms, eligibility criteria, data collection process, and results, ensuring a robust and reproducible research process.

The fourth chapter explores the practical application of AI tools in real-world scenarios through case studies and experimental setups. It evaluates the behavior of these tools in various company settings and analyzes the effectiveness of different types of AI tools (polyvalent vs. monovalent). It also discusses prompt engineering and its impact on AI performance.

The fifth chapter presents the findings from the data collected and experiments conducted. It includes detailed analyses of the data interpretation, a comparative study of the tools tested, and an evaluation of prompt effectiveness and construction, aiming to provide clear, actionable insights derived from the research.

The sixth chapter discusses the results in the context of the initial research questions and the broader field of AI and software development. It addresses the integration of AI in programming, explores the study's limitations, and suggests practical applications of the findings.

The seventh chapter summarizes the research's key findings, reflecting on the objectives set out at the beginning of the thesis. It discusses the research's practical implications for businesses and engineers, reflecting critically on the impact and utility of AI-driven code generation.

With its forward-looking perspective, the final chapter summarizes the research and sets the stage for future inquiry and development. It recommends specific areas for further study and highlights emerging trends that could significantly influence the development and application

of AI in software development. This chapter serves as a roadmap for ongoing exploration and advancement in the field.

Each chapter builds upon the previous, creating a comprehensive narrative that addresses the technical aspects of AI in code generation and considers ethical, practical, and future-oriented perspectives. This structure ensures that the thesis is a complete, informative, and forward-looking piece of research, contributing significantly to software development and AI.

2 Literature Review

This chapter is divided into two parts. First, the state of the art is meticulously explored, focusing on the transformative role of AI technologies such as ChatGPT and GitHub Copilot within various domains, including software development and programming education. The second part examines the ethical considerations, highlighting the critical need for responsible development and application of AI technologies.

This comprehensive review sets the stage for a deeper understanding of state of the art and the ethical considerations necessary for effectively and ethically integrating AI into future development strategies.

2.1 State of the Art

The rapid advancement of AI technologies has heralded a new era in various domains, fundamentally altering the landscape of software development, education, and intellectual property rights. This state-of-the-art section aims to meticulously explore and analyze AI's multifaceted impact, focusing on AI code generation tools like ChatGPT and GitHub Copilot and their implications in the contemporary realm of software development and programming education.

For that, the aim is to dissect the influence of AI tools on programming education, as exemplified by the study on how university programming instructors plan to adapt to AI code generation and explanation tools (Lau and Guo, 2023). This research includes assessing the pedagogical challenges and opportunities presented by such tools and understanding the emerging trends in educational strategies to integrate or counteract these AI advancements.

It will also be necessary to consider AI's increasing involvement in code generation, critically evaluating the quality, efficiency, and robustness of the code produced by tools like GitHub Copilot. This evaluation involves scrutinizing the methodologies employed in assessing AI-generated code and understanding the parameters that influence its effectiveness and reliability.

It is also essential to contemplate exploring the strategies and techniques, such as watermarking, developed to safeguard the intellectual property rights of AI-generated content, delving into AI's legal and ethical dimensions in development.

Furthermore, it is imperative to recognize the importance of security in software development and address the vulnerabilities and security challenges posed by AI-generated code by examining datasets like SecurityEval, designed to evaluate the safety and reliability of AI code generation techniques.

This document assesses the implications for software engineering practices by examining constraint-based code generation and the automation of code generation in Model-Driven Engineering (MDE) using machine learning to determine how AI is reshaping traditional software engineering practices and workflows.

This comprehensive analysis aims to provide a complete understanding of the current state of AI in development acceleration. It highlights technological advancements, educational implications, ethical considerations, and future potentials of AI tools in transforming the software development and education landscape.

2.1.1 Evaluating AI's Role in Programming Education

In the study titled "From 'Ban It Till We Understand It' to 'Resistance is Futile': How University Programming Instructors Plan to Adapt as More Students Use AI Code Generation and Explanation Tools such as ChatGPT and GitHub Copilot," researchers Sam Lau and Philip J. Guo conducted a seminal investigation into the impact of AI tools in programming education. This study presents an empirical snapshot of the educational landscape as it grapples with the emergence of AI tools like ChatGPT and GitHub Copilot.

The study undertaken by Lau and Guo adopted a qualitative research methodology. This research methodology involved conducting semi-structured interviews, a flexible and interactive form of interview that allows for in-depth exploration of participants' views. Twenty university instructors specializing in introductory programming courses were interviewed. These instructors were from nine different countries, providing a diverse international perspective on the impact of AI tools in programming education. This global perspective enriched the study, capturing various responses that reflect cultural and pedagogical diversities. The findings indicated that in the short term, many instructors planned to take immediate measures to discourage cheating. Opinions diverged regarding the long-term integration of these AI tools: one group of instructors wanted to ban them and continue teaching programming fundamentals, while another group aimed to integrate them into courses to prepare students for future jobs. These findings, along with the open research questions, aim to spur conversations about how to work with these tools in effective, equitable, and ethical ways (Lau and Guo, 2023a).

A related study, "Students' Perspective on AI Code Completion: Benefits and Challenges," conducted by Wannita Takerngsaksiri et al. from Monash University, offers valuable insights into students' experiences with AI code completion tools such as GitHub's Copilot and Amazon CodeWhisperer. This research introduced AutoAurora, an AI code completion tool integrated into the Visual Studio Code Extension, and investigated the benefits, challenges, and expectations from students' perspectives. Through interviews with ten participants, the study found that AI code completion enhanced students' productivity and efficiency by providing correct syntax suggestions, offering alternative solutions, and functioning as a coding tutor. However, it also highlighted concerns about over-reliance on AI tools, which may lead to a

surface-level understanding of programming concepts and diminished problem-solving skills (Takerngsaksiri et al., 2023).

The study's main findings reveal a bifurcation in the instructors' short-term and long-term strategies towards AI tools. In the short term, the primary concern centered around AI-assisted cheating, leading to immediate course adjustments, including a greater emphasis on exams and contemplation of outright bans on AI tools. Looking towards the future, opinions among educators diverged significantly. Some favored resisting the integration of AI tools, focusing instead on teaching programming fundamentals. Others leaned towards embracing these tools, seeing them essential to preparing students for future job markets where AI proficiency would be crucial (Lau and Guo, 2023b).

This study contributes significantly to the field by providing one of the first sets of empirical data on educators' reception and adaptation plans in response to AI's arrival in programming education. Its timing and relevance are particularly poignant, offering insights into educators' immediate reactions at a time when AI technologies in education are rapidly evolving. The global scope of the research underscores the widespread impact of AI tools in education, acknowledging that their influence transcends geographical and cultural boundaries.

However, the study has its limitations. The potential bias from self-selection among participants might have led to an overrepresentation of instructors with stronger opinions or more profound interest in AI tools. Furthermore, the discussions, albeit future-oriented, were anchored to the capabilities and implications of present-day tools like ChatGPT and GitHub Copilot, limiting the exploration of more innovative educational transformations. Additionally, the absence of student perspectives in the study leaves a gap in understanding the complete educational impact of these AI tools.

This study challenges traditional approaches in computing education and suggests a need for a continuous reevaluation of educational strategies considering technological advancements. It opens new research avenues, particularly in understanding the long-term impacts of AI on learning outcomes and developing educational strategies resilient to AI advancements. It emphasizes the necessity of an ongoing dialogue in the academic community, an adaptable approach, and a readiness to evolve alongside technological progress.

The inclusion of the student perspective, as highlighted in the Monash University study, underscores the importance of considering both educator and student viewpoints to understand the impact of AI tools in programming education. By addressing the benefits and challenges from both angles, future research can better inform the development of balanced and effective educational strategies that leverage AI advancements while maintaining robust learning outcomes.

2.1.2 Analyzing the Quality and Robustness of AI-Generated Code

Articles "How Important are Good Method Names in Neural Code Generation? A Model Robustness Perspective" and "Assessing the Quality of GitHub Copilot's Code Generation" collectively enhance our understanding of the quality and robustness of AI-generated code, contributing significantly to the field of AI as a development accelerator.

The first article delves into the significant impact of method names on the robustness of Pre-trained Code Generation Models (PCGMs). (Yang et al., 2023) introduces RADAR, comprising RADAR-Attack and RADAR-Defense components, to evaluate and bolster the robustness of PCGMs such as CodeGPT, PLBART, and CodeT5. RADAR-Attack underscores a critical vulnerability in PCGMs by altering method names, which disrupts the accuracy of code generation. This disruption in the accuracy demonstrates that even subtle changes in method names can lead to substantial accuracy reductions, highlighting the dependency of these models on method names for generating precise code. Conversely, RADAR-Defense counteracts this problem by synthesizing high-quality method names from functional descriptions, restoring performance. This innovative approach by Yang et al. reveals that method names are not just mere identifiers but play a pivotal role in the functionality and reliability of neural code generation. Their study accentuates the necessity of combining advanced AI algorithms with semantically clear and functionally descriptive method names to achieve robustness in AI-driven software development tools.

The second article empirically evaluates GitHub Copilot, an AI pair-programming tool that uses OpenAI's Codex model. This study assesses Copilot's code validity, correctness, and efficiency. It reveals Copilot's proficiency in generating valid code at a high success rate and illuminates its limitations, particularly in producing entirely correct code. The study balances acknowledging Copilot's potential as a development accelerator with critically examining areas for enhancement. Notably, it points out the need to improve the tool's comprehension of complex programming tasks and its ability to generate error-free code consistently (Yetistiren et al., 2022).

Adding to this, the article "Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT" by Yetistiren et al. provides a comparative analysis of three prominent AI code generation tools. This study examines the performance of GitHub Copilot, Amazon CodeWhisperer, and ChatGPT using code quality metrics such as Code Validity, Code Correctness, Code Security, Code Reliability, and Code Maintainability. The researchers found that the latest versions of ChatGPT, GitHub Copilot, and Amazon CodeWhisperer generate correct code 65.2%, 46.3%, and 31.1% of the time, respectively. They also noted improvement rates of 18% for GitHub Copilot and 7% for Amazon CodeWhisperer on their newer versions. The average technical debt, considering code smells, that each tool produced was 8.9 minutes for ChatGPT, 9.1 minutes for GitHub Copilot, and 5.6 minutes for Amazon CodeWhisperer. This comprehensive study highlights the strengths and weaknesses of each tool, providing valuable insights for practitioners and

contributing to the decision-making process in selecting the optimal tool for specific tasks (Yetiştirilen et al., 2023b).

These studies collectively demonstrate that while AI-generated code tools show promise in assisting and accelerating software development, there are notable limitations in their current capabilities. The first article focuses on the robustness of PCGMs, especially concerning method names. It provides valuable insights into how these tools can be misled or compromised, a crucial consideration for developers relying on AI for code generation. The strengths of GitHub Copilot, as highlighted in the second study, include its high success rate in generating valid code, which adheres to correct syntax and runtime requirements. Additionally, it demonstrates a notable capability in generating partially correct code, which, with minor modifications, can be made fully functional. However, the tool's limitations are evident in its handling of complex programming tasks. GitHub Copilot sometimes struggles to comprehend intricate coding requirements fully, leading to code that, while syntactically valid, may not fully achieve the intended task or could be inefficient regarding computational resource usage. The empirical study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT extends these findings by offering a comparative perspective on the performance of these tools, emphasizing areas for improvement in code reliability and maintainability.

These findings highlight the need for ongoing improvement in AI-driven code generation tools, particularly in enhancing their understanding of complex logic and optimizing code efficiency. Together, these articles contribute to the broader sense of AI in software engineering by highlighting both the potential and the challenges of AI-assisted code generation. They underscore the need for continued research and development to enhance AI-generated code's reliability, correctness, and robustness, ensuring these tools can support and accelerate the software development process more effectively.

2.1.3 Exploring Intellectual Property Concerns in AI-Generated Code

The research presented in "Protecting Intellectual Property of Large Language Model-Based Code Generation APIs via Watermarks" is a pivotal contribution to the field of AI, particularly in the arena of safeguarding intellectual property (IP) in the rapidly evolving landscape of significant language model-based code generation (LLCG). This study gains its significance from the context in which it is situated; the realm of AI is witnessing a surge in the deployment of sophisticated models like GitHub Copilot and OpenAI's Codex. These models, emblematic of substantial investments in data, computational resources, and intellectual effort, are vulnerable to imitation attacks, posing severe risks to their proprietary nature.

The core of the research revolves around ToSyn, a pioneering watermarking technique. This technique ingeniously embeds watermarks in the output of LLCG APIs by altering the distribution of token synonyms. The novelty lies in subtly tweaking the token distribution, embedding a unique identifier without hampering the model's utility for standard operations. ToSyn encompasses six distinct WM schemes, each harnessing different token synonyms. The

meticulous evaluation conducted using two renowned language models, CodeT5 and CodeBERT, demonstrates ToSyn's efficacy in watermark injection, ensuring high fidelity, robustness, and stealthiness. This methodological robustness extends to its strategy for IP verification and localizing malicious users, adding layers of security and traceability (Li et al., 2023).

While this research is groundbreaking, it has limitations. The concentration on Python as the programming language and the selection of specific LLCG models might restrict the broader applicability of the findings. The AI field is known for its rapid advancements, and the dynamic nature of imitation attacks may present unforeseen challenges to ToSyn. Although crucial, the study's focus on a defensive mechanism against IP theft is a singular aspect of the multifaceted AI development landscape.

This study is a significant leap in contributing to the broader understanding of AI as a development accelerator. It addresses the often-overlooked aspect of IP protection in AI, a factor that is vital for sustaining innovation and incentivizing the development of advanced AI models. By offering a viable solution to protect the intellectual labor invested in LLCG models, the study contributes to securing investments in AI and fosters a safer environment for further innovation. Therefore, this research enriches the existing body of knowledge and poses a challenge by introducing a new dimension to the conversation on AI development – IP protection and security. The introduction of ToSyn thus emerges as a crucial advancement, highlighting the need for continual evolution in strategies to protect AI's intellectual assets in a rapidly changing technological landscape.

2.1.4 Investigating Security Aspects in AI-Enabled Code Generation

The article "SecurityEval Dataset: Mining Vulnerability Examples to Evaluate Machine Learning-Based Code Generation Techniques" by Mohammed Latif Siddiq and Joanna C. S. Santos delves into the critical security aspect of AI-driven code generation. This study recognizes the growing reliance on machine learning for automated code generation and the inherent risks of such methods producing vulnerable or insecure code. The author's main contribution lies in developing the SecurityEval dataset, a curated collection of Python code samples linked to seventy-five vulnerabilities per the Common Weakness Enumeration (CWE). This dataset is tailored to evaluate the security robustness of machine learning-based code generation tools.

Methodologically, the SecurityEval dataset was constructed by mining examples of software vulnerabilities from reputable sources like CodeQL, the CWE list, Sonar Rules, and prior research. These examples were then formatted into prompts suitable for feeding into code generation models. The versatility of this dataset is displayed through its application in evaluating two distinct code generation models - the open-source InCoder and the closed-source GitHub Copilot. This evaluation is pivotal as it shifts the focus from just functional correctness to the security implications of the generated code. The authors employed manual

and automated strategies, including static analyzers like CodeQL and Bandit, to identify vulnerabilities in the code generated from the dataset's prompts (Siddiq and Santos, 2022).

The study's strengths lie in its practical approach to addressing a gap in evaluating AI-generated code, particularly regarding security. Creating a targeted dataset provides a valuable tool for developing and refining safer AI-driven code-generation techniques. However, this research has limitations. The SecurityEval dataset is currently Python-specific, which, while practical given Python's popularity, limits its applicability across other programming languages where different types of vulnerabilities might be more prevalent. Additionally, while thorough, the manual component of the evaluation process may need to scale better for larger datasets or more extensive code generation models.

In the context of AI as a development accelerator, this research underlines the critical need to balance speed and convenience with security considerations. While AI can significantly accelerate software development processes, there is an essential need for robust frameworks to assess and ensure the security of these AI systems' products. This study contributes to the existing body of knowledge by highlighting a critical aspect of AI in development – the need for security-aware evaluation methods. It challenges the current trajectory of AI code generation tools by shifting the focus from purely functional to security-conscious development, aligning with the growing awareness of the importance of cybersecurity in software development.

In summary, Siddiq and Santos' research is a significant step toward understanding and improving the security aspects of AI-enabled code generation. It provides a framework and resource for future studies and tool development in this rapidly evolving field.

A related study, "Evaluating the Impact of ChatGPT on Exercises of a Software Security Course," conducted by Jingyue Li et al. from NTNU, investigates the use of ChatGPT, specifically the GPT-4 model, in a university software security course. This study examines ChatGPT's ability to identify and fix vulnerabilities inserted in a web application used for course exercises. The researchers input vulnerable code into ChatGPT and measured its accuracy in identifying and correcting these vulnerabilities. They also assessed whether ChatGPT could provide appropriate sources to support its outputs (Li et al., 2024).

The study found that ChatGPT could identify twenty of the twenty-eight vulnerabilities inserted into the web application, reported three false positives, and found four additional vulnerabilities that were not intentionally inserted. Additionally, ChatGPT made nine satisfactory breach testing and fixing recommendations for ten specified vulnerabilities and often pointed to related sources of information.

This research highlights several key findings. ChatGPT demonstrated substantial accuracy in identifying vulnerabilities, though some cross-file vulnerabilities were limited due to token limits. It provided satisfactory breach testing methods and fix recommendations for most vulnerabilities and often supplied adequate sources to support its findings, although some were outdated.

The study underscores ChatGPT's potential to impact software security education by enabling students to identify and fix vulnerabilities effectively. However, it also raises concerns about students' over-reliance on AI tools, which could undermine the learning of fundamental skills and thorough understanding.

In summary, both Siddiq and Santos' and Li et al.'s research contribute significantly to the field by addressing the security implications of AI in software development. Siddiq and Santos provide a framework for evaluating the security of AI-generated code. Li et al. explore the educational impact of AI tools like ChatGPT in teaching software security. Both studies emphasize the need for security-aware approaches in AI-driven software development and education, highlighting the importance of balancing automation with robust security practices and thorough learning experiences.

2.1.5 Assessing the Implications for Software Engineering Practices

The research articles "Constraint-based Code Generation" and "Automating Code Generation for MDE Using Machine Learning" delve into innovative approaches to automating and enhancing the code generation process, albeit with different methodologies and contexts.

"Constraint-based Code Generation" by Castañeda Lozano et al. introduces a novel approach utilizing constraint programming for generating assembly code, focusing on the three primary tasks of instruction selection, register allocation, and instruction scheduling. This study is revolutionary in its use of constraint models and solving techniques, which starkly contrasts the traditional staged and heuristic algorithm-based methods in compiler back-ends. The approach promises simplicity, flexibility, robustness, and the potential for optimal code generation. However, its effectiveness is mainly showcased in more straightforward instruction set architectures like MIPS32, and the real challenge lies in applying this method to more complex and irregular architectures, especially those familiar with embedded systems (Lozano et al., 2013).

In contrast, Xue's study on "Automating Code Generation for MDE Using Machine Learning" explores the use of symbolic machine learning, precisely a method termed Code Generation by Example (CGBE), to automate the creation of code generators in the Model-Driven Engineering (MDE) domain. This approach significantly differs from traditional Model-to-Model, Model-to-Text, and Text-to-Text methods in MDE, offering a more streamlined process with less reliance on extensive manual effort and expertise. The CGBE method, while demonstrating efficient performance in learning UML-to-Java code generators, still needs expansion to accommodate various programming languages and optimize generated code. One of its strengths is its ability to produce explicit and inspectable rule sets, a clear advantage over non-symbolic ML methods requiring large datasets and opaque training processes (Xue, 2023).

Both studies contribute significantly to the evolving landscape of AI in software development. While Castañeda Lozano et al.'s constraint-based approach pushes the boundaries of

traditional code generation techniques, offering a glimpse into generating optimal code, Xue's research demonstrates the practicality and agility of symbolic machine learning in automating the construction of code generators, potentially revolutionizing MDE practices. However, both approaches face challenges in scaling and adapting to diverse and complex programming scenarios. The success of these methodologies in broader applications remains to be seen, necessitating further research and development.

A related study, "The Impact of AI on Developer Productivity: Evidence from GitHub Copilot," conducted by Sida Peng et al., investigates the productivity effects of GitHub Copilot, an AI pair programmer, in a controlled experiment. In this study, software developers were tasked with implementing an HTTP server in JavaScript as quickly as possible. The treatment group, which had access to GitHub Copilot, completed the task 55.8% faster than the control group, which did not have access to the AI tool. The findings suggest significant productivity gains, particularly for less experienced developers, older programmers, and those with higher daily coding hours. This research underscores the potential of AI pair programmers to support transitions into software development careers and highlights the broader economic implications of AI-driven productivity improvements (Peng et al., 2024).

In summary, while Castañeda Lozano et al.'s constraint-based approach and Xue's symbolic machine learning method represent significant advancements in automated code generation, Peng et al.'s investigation into GitHub Copilot illustrates the practical productivity benefits of AI tools in real-world programming tasks. Each study emphasizes different aspects of AI's impact on software development, collectively contributing to a deeper understanding of how AI can transform this field.

2.2 Ethical Considerations

The ethical implications of utilizing AI as a development accelerator are vital, significantly influencing this technological approach's integrity, social impact, and effectiveness. This section explores these ethical considerations, focusing on the responsible development and application of AI technologies in various developmental contexts.

Adherence to established ethical standards and guidelines is crucial. This research aligns with the principles set forth by prominent bodies such as the Institute of Electrical and Electronics Engineers (IEEE) Code of Ethics ("IEEE Code of Ethics," 2020), as well as relevant data protection regulations like the General Data Protection Regulation (GDPR) ("Regulation 2016/679 - RGPD," 2016).

Firstly, the application of AI in development projects, whether in infrastructure, healthcare, or education, raises significant ethical questions regarding decision-making transparency, bias, and accountability. It is essential to ensure that AI systems are designed and deployed to promote fairness and inclusivity, avoiding perpetuating existing social or economic inequalities.

Secondly, the rapid development and implementation of AI technologies necessitate carefully considering intellectual property rights and licensing agreements. These considerations include the responsible use of open-source frameworks and proprietary AI models, ensuring compliance with legal and ethical standards.

Integrating AI in development projects introduces complex data privacy and security challenges. Safeguarding sensitive information, particularly in sectors like healthcare and finance, is critical. Compliance with international standards such as ISO/IEC 27001:2022 (2022) for information security management is imperative to protect data integrity and privacy.

In conclusion, while AI presents significant opportunities for accelerating development processes, it is crucial to navigate these ethical challenges carefully to maximize its positive impact while mitigating potential risks.

2.2.1 Celfocus Ethics Regarding Generative AI

As a forward-thinking technology company, Celfocus recognizes the transformative potential of Generative AI while understanding the importance of ethical considerations in its deployment. The commitment to responsible AI usage is reflected in these moral guidelines, aiming to align AI applications with core values and societal norms. These principles guide the ethical use of Generative AI at Celfocus, ensuring that technologies contribute positively to society and uphold human dignity, fairness, and privacy.

Celfocus deploys AI technologies with a thorough understanding of their ethical implications, ensuring they align with the company's societal values and mission. Respect for human dignity is paramount; AI technologies developed and used by Celfocus will never be used in ways that demean, dehumanize, or infringe upon human rights. Any usage that is discriminatory, defamatory, or harassing is strictly forbidden.

Non-discrimination is a core principle, and AI systems are designed and operated to prevent any form of bias related to race, gender, sexual orientation, age, disability, or other personal characteristics. Celfocus aims for AI applications to benefit all people equally.

Privacy and data protection are also prioritized. Compliance with laws such as GDPR is mandatory, and users must anonymize personal data, maintain transparency in data usage, and keep access credentials confidential. Confidentiality is critical, with strict measures to ensure that GenAI tools are not used to input or upload confidential, proprietary, or sensitive company information. This helps prevent breaches of confidentiality, reduces the risk of disclosure, and protects the company's rights and competitive standing.

Transparency and explainability are emphasized, with AI systems' decision-making processes being as clear as possible. Users must clearly explain the inputs and critically judge AI outputs, validating information before application. Accountability is a fundamental tenet, with Celfocus holding responsibility for the outcomes of AI systems and their usage. AI must not be used to

generate or spread false or misleading information, and all AI usage should adhere to Celfocus's values and Code of Conduct.

Reliability and safety are crucial, and AI systems are developed to operate reliably and safely. Users are responsible for ensuring these systems behave as intended and do not cause harm. The company's learning platform provides training on the correct and ethical use of AI, ensuring all users are well-informed on best practices.

Finally, all AI practices must comply with the legal and regulatory frameworks applicable in the areas where Celfocus operates. This comprehensive approach to the ethical use of Generative AI underscores Celfocus's dedication to responsible innovation and the positive contribution of technology to society.

3 Methodology

This chapter is divided into two parts. The first part provides a detailed description of the research process, outlining the structured and systematic approach used to investigate the role of AI as a Development Accelerator. The second part illustrates how the various AI tools will be evaluated. This will be done by determining the criteria and methods that will assess the AI tools.

3.1 Research Process

This research investigates the role of AI as a Development Accelerator, using the PRISMA method to ensure a structured and systematic approach. The PRISMA framework, known for its rigor in systematic reviews and meta-analyses, provides a transparent and replicable methodology, aligning perfectly with the study's aim to assess the impact of AI technologies in development initiatives comprehensively.

3.1.1 Research Questions

In exploring the multifaceted impact of AI and machine learning in the acceleration of software development, this thesis is guided by the following research questions, each addressing an essential key aspect of AI's integration into this field:

RQ1. How are AI and machine learning transforming the landscape of programming education and code generation?

RQ2. What are the implications of AI-generated code regarding quality, robustness, and security?

RQ3. How does AI influence intellectual property rights and legal considerations in software development?

RQ4. How is AI reshaping traditional software engineering practices?

The first question explores AI's expanding influence in educational settings. The goal is to uncover the changes AI is bringing to pedagogical approaches and its potential to reshape the educational landscape for programming. The second question focuses on the practical aspects of code generation, delving into the efficacy and safety of AI-produced code. The third question investigates the complex interplay between AI and intellectual property rights within software development. The last question seeks to understand AI's impact on the methodologies and practices of software engineering.

3.1.2 Data Sources

The first step is identifying the data sources for the systematic review. Table 1 shows the selected electronic databases.

Table 1 - Data sources

Identifier	Database	URL
DS1	Science Direct	https://www.sciencedirect.com
DS2	ACM Digital Library	https://dl.acm.org/
DS3	IEEE Xplorer	https://ieeexplore.ieee.org/Xplore/home.jsp
DS4	ArXiv	https://arxiv.org/

These databases contain relevant peer-reviewed academic publications, and they were chosen because they are considered highly appropriate by researchers in information technology.

3.1.3 Search Terms

The keywords for the search query are Artificial Intelligence, AI, Machine Learning, Software Development, Code Generation, and Automation in Coding.

Search queries were created using the keywords to cover the multiple facets of the topic in hand:

(Artificial Intelligence OR AI OR Machine Learning) AND (Software Development OR Code Generation OR Automation in Coding).

(ChatGPT OR GPT OR Copilot OR GitHub Copilot OR TabNine) AND (Benefits OR Challenges OR Impact OR Code Quality OR Security)

By combining these terms using the OR operator with the AND operator, the first search query only returns articles that contain at least one of the specified AI-related terms, are related to code generation, are focused on the Java language, and have comparisons between AI technologies. The other search query only returns articles that contain one of the AI tools and focuses on its benefits, challenges, impact, code quality, or security.

3.1.4 Eligibility Criteria

The inclusion criteria for an article to be selected were:

IC1. The source pertains to the field of AI as a Development Accelerator.

IC2. The source is peer-reviewed.

IC3. The source studies scenarios of the application of AI in development with specific use cases.

The exclusion criteria for an article to be discarded were:

EC1. The source does not provide practical scenarios/applications of AI in development.

EC2. The source has not been published in English.

3.1.5 Data Collection Process

The PRISMA systematic review process was followed to obtain relevant articles, as seen in Figure 1. This process consists of three steps: Identification involves getting all articles from data sources that meet the eligibility criteria. A total of 270 valid articles were identified. Screening consisted of two parts; the first was an abstract screening of all identified articles, marking them as “Relevant,” “Irrelevant,” and “Possibly Relevant.” “Possibly Relevant” articles were screened again to be classified as “Relevant” or “Irrelevant.” Twenty-four articles were deemed “Relevant.” The relevant articles were thoroughly analyzed in the second part to find information fitting the research questions. Seven articles were considered to answer at least one research question.

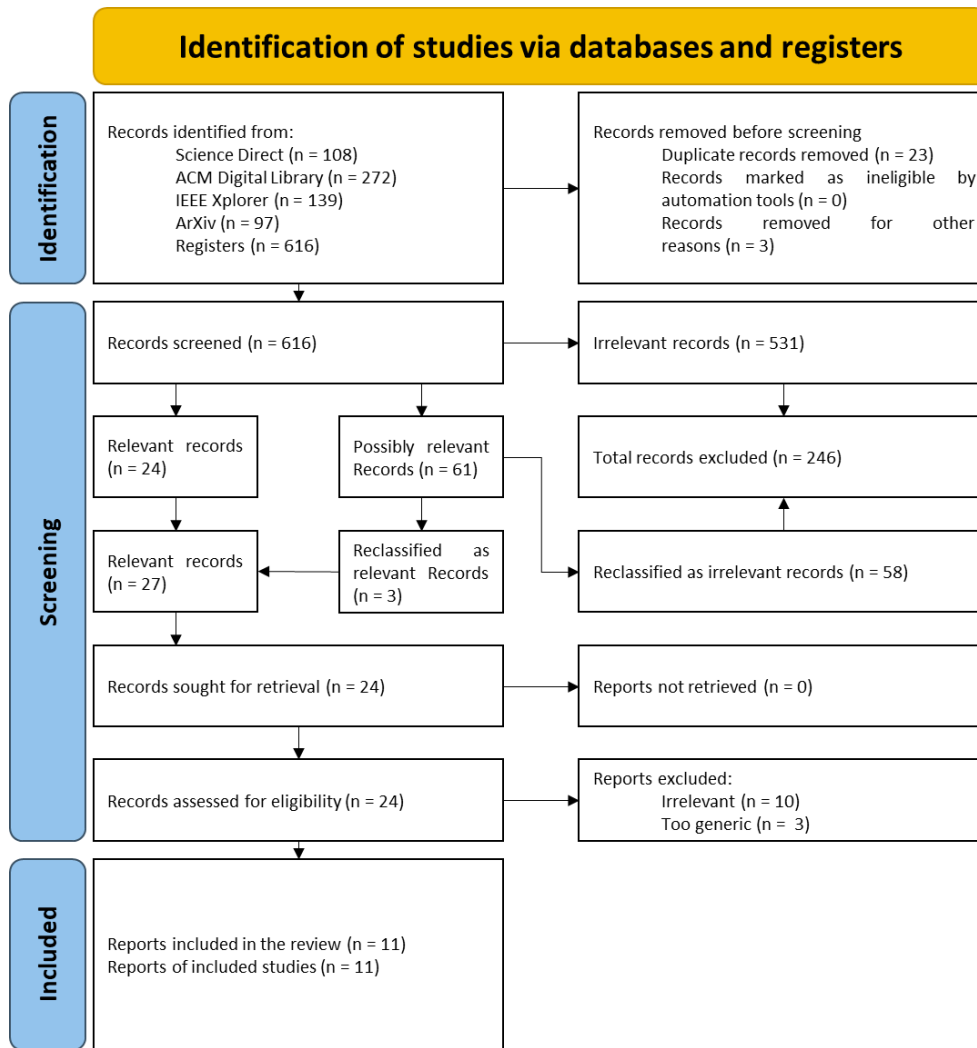


Figure 1 Flowchart of the Data Collection Process

3.1.6 Results

RQ1. How are AI and machine learning transforming the landscape of programming education and code generation?

Of eleven retrieved search results, two provide insights into the transformative effects of AI and machine learning in programming education and code generation. The works are (Lau and Guo, 2023b) and (Takerngsaksiri et al., 2023).

RQ2. What are the implications of AI-generated code regarding quality, robustness, and security?

Of eleven retrieved search results, five delved into the implications of AI-generated code regarding its quality, robustness, and security. The works are (Yang et al., 2023), (Yetistiren et al., 2022), (Siddiq and Santos, 2022), (Yetiştirten et al., 2023b), and (Li et al., 2024).

RQ3. How does AI influence intellectual property rights and legal considerations in software development?

Of eleven retrieved search results, one addresses the influence of AI on intellectual property rights and legal considerations in software development. The work is (Li et al., 2023).

RQ4. How is AI reshaping traditional software engineering practices?

Of eleven retrieved search results, three examine how AI reshapes traditional software engineering practices. The works are (Lozano et al., 2013), (Xue, 2023), and (Peng et al., 2024).

3.2 Tool Comparison Criteria

This section provides an overview of the methodology for assessing AI-driven code generation tools in today's demanding software development environments. The evaluation framework encompasses essential areas such as performance efficiency, user experience, integration capability, cost-effectiveness, scalability, security, and reliability. A thorough evaluation will be carried out to meticulously assess the selected tools.

Performance Efficiency

Testing scenarios will be crafted to measure the speed and accuracy of code generation. Metrics such as the time taken to generate code and the number of errors or issues in the code compared to a human-generated baseline will be used.

User Experience

Before granting access to the GitHub Copilot license, the Celfocus developers were surveyed about AI tools to gather qualitative data on user satisfaction, ease of use, thoughts on the theme, and the learning curve associated with each tool.

Integration Capability

This assessment will include version compatibility tests with different IDEs and operating systems to ensure seamless workflow transitions. API connectivity and plugin support will be examined to gauge how well the tool integrates with external software and data sources. The ease with which the tool can be embedded into existing CI/CD pipelines will also be crucial, especially for teams relying on automated build and deploy processes.

Cost-effectiveness

A detailed cost-benefit analysis will be performed, considering direct costs, such as purchase and operational expenses, and indirect costs, like user training and downtime during implementation, and weighing them against the benefits, such as reduced coding time and decreased need for debugging.

Scalability

This criterion will be assessed by observing the tools' performance in handling large-scale projects or increasing demands, examining factors such as processing speed and system resource utilization.

Security and Reliability

Stress tests will evaluate each tool's robustness under various conditions. Historical data reviews on known security issues or downtime incidents will also be made.

These criteria will guide the comprehensive comparison of the tools in the next section, identifying the best fit for different development needs and environments. This approach ensures a balanced evaluation, focusing on each tool's strengths and limitations and providing a clear direction for the report.

4 Case Studies and Experimentation

This chapter explores the practical application of AI-driven code generation tools, presenting an in-depth analysis through case studies and experimental evaluations. These investigations demonstrate the tools' operational capabilities and limitations within varied real-world software development environments. Each case study and experimental setup is meticulously designed to assess how these AI technologies perform across different dimensions, efficiency, adaptability, and impact on the software development lifecycle. By exploring the operational dynamics of the tools in company settings, this chapter seeks to uncover behavioral patterns and measure the tangible benefits and drawbacks of integrating AI into coding practices. Additionally, these tools' concepts of polyvalence versus monovalence will be explored to determine their versatility and specialization, contributing to a nuanced understanding of their suitability for diverse development tasks. Lastly, an exploration of prompt engineering will highlight how effectively AI tools understand and execute complex coding tasks based on the nuances of user input, providing insights into their practical effectiveness and potential limitations.

4.1 Tools Used

This section highlights the potential benefits of several AI-driven code-generation tools relevant to Celfocus's objectives in enhancing the software development process. The evaluation focuses on each tool's operational functionalities and strategic alignment, assessing its potential to significantly improve productivity, integration, and code quality within our development workflows.

Developed by GitHub and OpenAI, GitHub Copilot stands out for its ability to suggest code snippets directly within popular IDEs like Visual Studio Code. It utilizes OpenAI's advanced GPT models to predict and generate code as the user types ("GitHub Copilot overview," 2024). The tool has recently incorporated features like Copilot Chat and automated pull request reviews, which provide real-time coding advice and automated code testing suggestions ("Code faster and better with GitHub Copilot's new features in Visual Studio," 2024). Celfocus is particularly interested in GitHub Copilot due to its widespread adoption and potential to enhance developer productivity significantly. The evaluation will focus on its integration within Celfocus's development workflow and its efficacy in improving coding efficiency (Wilson-Thomas, 2023).

A direct competitor to GitHub Copilot, TabNine supports many programming languages and integrates seamlessly with multiple IDEs. It predicts and suggests the following parts of code based on the already-typed text using machine-learning algorithms (Valerie, 2023). TabNine will be evaluated to compare its features against GitHub Copilot, aiming to determine which tool offers better support and efficiency in real-world development scenarios at Celfocus.

While both CodeWhisperer and Snyk Code offer significant potential benefits—CodeWhisperer in real-time code recommendations (“Features - CodeWhisperer,” n.d.) and Snyk Code to enhance code security through vulnerability scans (“Snyk Code | Code Security Analysis and Fixes - Developer First SAST,” n.d.)—they will not be included in this study. Celfocus has not expressed interest in the specific functionalities of integrating cloud services or conducting deep security analyses at this stage, preferring to focus on direct coding assistance tools.

ChatGPT 3.5, ChatGPT 4.0, and ChatGPT 4o iterations of OpenAI's models will be assessed to explore the advanced capabilities of ChatGPT 4.0 (Maheshwari, 2023) and ChatGPT 4o (OpenAI, 2024) when compared to its predecessor while focusing on whether the enhanced features justify its cost for Celfocus. The models' ability to generate accurate and efficient code makes them relevant for evaluating improvements in natural language understanding and generation that could benefit Celfocus's technical and professional applications.

Custom GPTs are specialized versions of the GPT model designed to address specific tasks or industries, providing tailored responses that enhance performance in specialized domains (Lim et al., 2021). The study will consider these models to see if the benefits of custom solutions offer a significant advantage over general models like ChatGPT 4.0 regarding cost-effectiveness and performance in specialized areas.

IntelliJ IDEA and Visual Studio Code are the primary platforms for testing these tools. These platforms are not only for being the ones used in Celfocus, but also for their wide recognition and robust feature sets, which make them excellent choices for assessing integration and effectiveness across different coding environments. Additionally, browsers like Brave and Chrome will be used to test the capabilities of ChatGPT models in web-based code consultations.

In conclusion, the selection of tools for this evaluation covers a comprehensive range of AI-driven code assistance applications, from real-time coding suggestions to advanced model comparisons. This tailored approach ensures that the assessment aligns closely with Celfocus's immediate needs and priorities, providing reassurance of its relevance to our operations.

4.2 Behavioral Patterns and Predictions in Companies

Integrating AI completion tools like GitHub Copilot, TabNine, CodeWhisperer, Snyk Code, and ChatGPT into company workflows is critical for how technology impacts workplace dynamics and productivity. Whether these tools lead companies down a "Dummy path" or serve as "Accelerators" depends on how they are implemented and managed within the corporate culture and operational framework.

For several reasons, companies' future utilizing AI tools will skew more toward the "Accelerator" path. AI tools can significantly reduce the time spent on routine and repetitive tasks, allowing employees to focus on more complex and creative work. This can increase

productivity as human expertise is leveraged for higher-level decision-making and problem-solving. A study from GitHub confirms that developers using Copilot experienced a notable increase in productivity and satisfaction, suggesting that AI can significantly accelerate software development processes (Peng et al., 2024).

AI tools, especially in coding and software development, often learn from new data and adapt over time. This feature continually refines and improves the tools' suggestions, helping developers stay on the cutting edge of programming techniques and technologies. Research suggests that AI-assisted tools like GitHub Copilot, Amazon CodeWhisperer, and ChatGPT have been shown to improve developer productivity by generating code from natural language prompts or partial code inputs, which can be integrated into the software more efficiently (Yetiştirilen et al., 2023c).

The most effective use of AI tools is with human oversight. The concept of "pair programming" with AI, where the tool suggests improvements or alternatives, mirrors the human pair programming approach and ensures that human developers review and refine AI-generated code. This combination can enhance the output quality and the human programmers' skills. For instance, Copilot is reported to improve the overall coding experience by reducing keystrokes and helping developers recall syntax, which enhances their engagement and productivity (Liang et al., 2023).

Companies increasingly recognize the importance of upskilling their employees to work effectively with AI. Training programs focusing on the synergistic use of AI tools can prevent the "Dummy path" scenario where reliance on AI leads to skill degradation. Instead, they promote a culture where AI is a tool that enhances human capabilities rather than replacing them. Successful AI integration involves strategic upskilling and reskilling programs. Companies that adopt comprehensive AI strategies, including employee training and development, are better positioned to leverage AI's full potential (Davenport and Mittal, 2022).

AI tools can analyze vast amounts of data and identify patterns that might not be visible to human analysts. This capability can lead to innovative solutions and improvements in various business processes, from code development to market analysis. For example, AI models combining pattern recognition with machine learning, such as the PRML model, improve stock trading decisions by identifying profitable patterns and making accurate forecast (Lin et al., 2021).

However, the trajectory towards the "Accelerator" path is not guaranteed. It requires thoughtful implementation, continuous training, and a balanced approach to ensure AI tools enhance rather than overshadow human skills. The "Dummy path" risk remains, particularly in environments with excessive reliance on automation and inadequate attention to developing human expertise alongside AI capabilities. Therefore, the future will likely see a mixed landscape where the success of AI integration will vary based on company policies, culture, and the continuous evolution of AI technologies.

4.3 Polyvalence vs Monovalence in Tools

In AI, tools can be classified based on their ability to perform various tasks. Polyvalent tools are theoretical concepts and practical solutions that can handle multiple functions, displaying versatility across applications and environments. Conversely, monovalent tools are designed to perform a singular, specific function with a high degree of specialization. Understanding the distinction between these classifications is crucial as it helps users choose the right tool for their needs, ensuring optimal performance and efficiency.

GitHub Copilot is considered a polyvalent tool due to its versatility in software development. It integrates directly with the user's coding environment and offers more than just code completion; it provides contextually relevant code examples, entire code snippets, and even complete functions based on natural language prompts and comments within the code. This tool is designed to assist in various programming languages and across different frameworks, adapting to the user's style and requirements. Beyond just writing code, it helps debug and understand legacy code and can generate unit tests, making it multifunctional within its domain (Zhang et al., 2023).

TabNine is another example of a polyvalent tool, although its focus is narrower than that of GitHub Copilot. It offers a deep learning-powered code completion feature that works across all programming languages supported by the IDEs it integrates. TabNine predicts not only the following few characters or words but entire lines of code. Its ability to adapt to the user's coding style and preferences and provide relevant suggestions based on the current code context allows it to serve multiple aspects of coding beyond simple completion, including reducing bugs and improving code quality (Vaithilingam et al., 2023).

Due to its broad range of capabilities, ChatGPT is the epitome of a polyvalent AI tool. Initially designed to generate human-like text based on the prompts it receives, its applications extend far beyond simple text generation. ChatGPT can assist with answering questions, composing emails, creating content, tutoring on various topics, translating languages, and even simulating conversational partners for practice. Its versatility makes it applicable in educational, creative, technical, and casual domains, showcasing a wide range of functionalities not limited to a single purpose (Sridhara et al., 2023).

Custom GPTs (Generative Pre-trained Transformers) tailored for specific tasks or industries also exhibit polyvalence, depending on their design and training. These models can be fine-tuned to perform various functions within their specialized areas. For example, a Custom GPT developed for the medical field might assist with generating diagnostic reports, answering patient questions, and providing training scenarios for medical students (McDermott et al., 2023). Similarly, a Custom GPT for customer service might handle inquiries, offer support, and even upsell products based on customer interactions (Saka et al., 2024). The ability to

customize these models for specific purposes while retaining the ability to perform multiple tasks within those realms underscores their polyvalence.

The exploration of GitHub Copilot, TabNine, ChatGPT, and Custom GPTs illustrates the significant advantages of polyvalent AI tools in modern technology landscapes. These tools enhance productivity and efficiency and offer flexibility and adaptability across various tasks and industries. By understanding the capabilities and applications of these AI solutions, users and developers can better leverage their potential to meet diverse needs, fostering innovation and driving progress in numerous fields. The trend towards polyvalence in AI tools is a testament to AI's evolving complexity and expanding capabilities, promising a future where versatility in digital tools becomes a fundamental criterion for technological development.

4.4 Prompt Engineering Analysis

Prompt Engineering is an increasingly crucial field in AI, particularly as AI technologies become more advanced and integrated into various aspects of daily life and professional fields. This detailed exploration aims to understand the nuances of prompt engineering, how it shapes interactions with AI models, and its pivotal role across different AI-driven applications.

Prompt Engineering involves crafting queries or inputs fed into AI models to effectively guide their processes and output. It is especially relevant in systems based on machine learning, where the quality and specificity of inputs can significantly influence the accuracy and utility of outputs. Prompt engineering involves translating a user's vague or complex requirements into a clear, concise, and direct query that an AI system can understand and respond to efficiently (Clariso and Cabot, 2023).

AI models, particularly those based on transformer architecture like GPT (Generative Pre-trained Transformer), rely heavily on the data they were trained in. These models generate outputs based on patterns learned during their training phase. As such, prompt engineering's role is to provide input and frame it to align with these learned patterns to elicit the most relevant and accurate responses.

The prompt engineering method requires understanding the AI model's capabilities and constraints. It combines technical knowledge about the model's workings and creative skills to manipulate the input to fit these parameters.

Prompt Engineering is not limited to any single application. Still, it spans various domains, from NLP (Natural Language Processing) to computer vision and even emerging fields like AI in healthcare and autonomous driving.

- **Natural Language Processing (NLP):** In NLP, prompt direct conversation models like chatbots help AI systems understand context in language translation or extract specific

data from large text corpora. Effective, prompt engineering can mean the difference between an AI understanding an ambiguity in a sentence or missing the subtlety entirely.

- **Computer Vision:** In image generation or analysis, how a prompt is structured can determine whether an AI model recognizes specific objects in images or generates images that meet the user's expectations. For instance, AI used in medical imaging can be prompted to identify anomalies in X-rays or MRIs with high precision, which is crucial for accurate diagnostics.
- **Code Generation and Software Development:** Tools like GitHub Copilot and TabNine rely on well-engineered prompts, such as code comments or partial codes, to generate accurate and usable code suggestions. These prompts can accelerate development processes and reduce developers' cognitive load.

Several modern AI tools utilize prompt engineering extensively:

- **ChatGPT by OpenAI:** ChatGPT interacts with users through typed prompts as a text-based model. The clarity and detail of these prompts directly influence the quality of the response. For example, a prompt asking for a "summary of a scientific paper" should ideally include specifics about the paper's focus, the desired length of the summary, and any particular emphasis (like methodology or results).
- **GitHub Copilot:** This tool assists programmers by providing code suggestions based on their comments or code. Prompt engineering is how effectively a programmer can communicate intent through comments or code snippets, guiding the AI to produce the most relevant and contextually appropriate code.
- **TabNine:** Like GitHub Copilot, TabNine offers completions using the context provided by the existing code. The initial code or comments act as prompts, and their precision can significantly enhance the relevance of the suggestions made by TabNine.

While prompt engineering has proven effective across many domains, it is not without its challenges:

- **Complexity and Accessibility:** Crafting effective prompts requires a deep understanding of the AI model's training and capabilities, which can be a barrier for non-expert users. There is a growing need for more intuitive systems that require less specialized knowledge to interact effectively with AI.
- **Bias and Ethics:** Since AI models learn from vast datasets that may contain biases, the prompts can inadvertently perpetuate or amplify these biases. Engineers must be vigilant about the data sources and the potential ethical implications of their prompts.
- **Adaptability and Learning:** Ideally, AI systems should respond to prompts and learn from them, adapting over time to provide better and more personalized responses. Presents a frontier for research, pushing toward AI systems that evolve based on user interaction.

Prompt engineering is a sophisticated and essential aspect of working with AI. As AI technologies permeate various sectors, effectively communicating with these systems becomes increasingly critical. The evolution of prompt engineering promises enhanced

interactions and broader accessibility, opening new possibilities for AI applications that are more intuitive, efficient, and equitable. In everyday technology, complex software development, or critical healthcare diagnostics, efficient engineering will undoubtedly be essential.

5 Results

This chapter presents the findings from a comprehensive evaluation of AI-driven code generation tools and methodologies. This chapter is structured to provide a clear and detailed account of the data collection process, interpretation of the insights derived from the data, comparative analysis of the tools under study, and assessment of the effectiveness of prompt engineering techniques. The aim is to offer a robust understanding of how these tools perform in practical settings, their relative strengths and weaknesses, and the impact of prompt design on their functionality. Through a systematic presentation of results, this chapter seeks to contribute valuable insights into the viability and effectiveness of integrating AI technologies into modern software development workflows.

5.1 Data Collection Procedure

The data collection phase of this thesis involved a systematic approach to gathering relevant information from multiple sources to ensure a comprehensive analysis of AI-driven code generation tools. This section details the processes and methodologies employed to collect the necessary data, emphasizing the rigor and thoroughness of the approach.

5.1.1 Data Sources

The primary data sources for this research were diverse and comprehensive, ensuring a robust dataset to evaluate AI tools effectively.

Science Direct and the ACM Digital Library were accessed to retrieve peer-reviewed articles related to AI and software development. These databases provided a wealth of scholarly resources that informed an understanding of current advancements and challenges in AI-driven code generation.

Several AI tools were focused on to understand their capabilities and limitations:

- **GitHub Copilot:** Data was collected through hands-on experiments and practical usage within development environments.
- **TabNine:** Information was gathered via direct usage, evaluating the tool's performance in real-world coding scenarios.
- **ChatGPT (versions 3.5, 4.0, 4o, and Custom GPTs):** These iterations were evaluated through practical application scenarios and prompt effectiveness studies, providing insights into their improvements and applicability.

A survey was conducted with Celfocus developers who have experience using AI tools in software development. The results provided qualitative insights into the tools' performance, ease of integration, and overall usability within the company's development workflows.

The analysis focused on code repositories and projects worked on at Celfocus. This included examining commit histories, code quality, and the frequency of manual interventions required to refine AI-generated code. This approach ensured the data was directly relevant and practical for the study's objectives.

5.1.2 Evaluating AI Tools for Code Generation

To ensure the reliability and validity of the collected data, a variety of methods were employed, combining both quantitative and qualitative approaches.

A systematic review of existing literature on AI-driven code generation was conducted. This involved identifying, selecting, and synthesizing research articles that met predefined inclusion criteria. The literature review helped frame the research context and provided a theoretical foundation for evaluating AI tools.

Practical experiments were conducted using GitHub Copilot, TabNine, and ChatGPT. These experiments were designed to test each tool's ability to generate functional code for Java microservices within the Spring Boot framework. Specific tasks included:

- Implementing tests (Tests for simple methods like get by, create, and update)
- Developing APIs (Moderate APIs like getting by with many parameters and creating an object with many connections to others)
- Altering Swagger documentation (From simple modifications of adding a parameter to a definition to creating a new API from scratch)
- Configuring Jenkins pipelines (Modifying Jenkins pipeline to communicate with other Microservices by calling it from another Microservice)
- Integrating database connections (Integrating database connections to MongoDB and PostgreSQL)
- Improving existing code (Refactoring code that is not understandable and improving it, making it better and easier to understand)

Metrics such as code accuracy, time efficiency, and the need for manual adjustments were meticulously recorded.

Structured surveys were distributed to developers at Celfocus. The surveys included questions about the tools' ease of use, workflow integration, and overall satisfaction. This feedback provided a practical perspective on the tools' performance and usability within a corporate environment.

The code generated by AI tools in real-world projects, specifically those worked on at Celfocus, was examined. This involved reviewing the quality, readability, and maintainability of the code. Special attention was paid to common issues such as code errors, inefficiencies, and security vulnerabilities, ensuring the analysis was directly relevant to practical applications.

5.1.3 Data Collection Challenges

Several challenges were encountered during the data collection process, each addressed to maintain the study's integrity.

Some AI tools required licenses or subscriptions, which posed logistical and financial constraints. However, trial versions and academic access were leveraged to ensure comprehensive coverage.

Ensuring unbiased responses from survey participants was challenging. Efforts were made to include diverse participants with varying experience levels and organizational backgrounds within Celfocus. This diversity helped mitigate potential biases and provided a balanced view of the tools' performance.

Managing the large volume of data collected from different sources and ensuring its relevance to the research objectives required meticulous organization and filtering processes. This was crucial in maintaining the study's focus and applicability.

5.1.4 Data Collection Summary

The data collected provided a robust foundation for analyzing the effectiveness and practicality of AI-driven code-generation tools. This summary highlights the key data points and introduces the framework for understanding how these data points will be utilized in the subsequent analysis. It is essential to introduce the methodology for data evaluation here to ensure clarity and cohesion when presenting the findings.

The collected data encompasses quantitative metrics and qualitative insights, collectively informing the comprehensive evaluation of the AI tools. Specifically, the key data points include measuring code accuracy rates, capturing the efficiency of the tools in generating code, and comparing to human-generated code baselines. Additionally, the data includes the frequency and types of manual adjustments needed, analyzing how often and what kind of manual interventions were required to refine the AI-generated code.

Qualitative insights were gathered through surveys and interviews with developers at Celfocus, offering practical perspectives on the tools' performance and ease of use. These insights are complemented by detailed case studies highlighting specific use cases and outcomes, illustrating the practical applications and challenges encountered. Furthermore, the code quality analysis focused on readability, maintainability, and adherence to standards, assessing

the overall quality of the code generated by the AI tools. This analysis also identified common errors and vulnerabilities, informing the robustness and reliability of the tools.

Performance efficiency was evaluated by measuring the speed and accuracy of code generation through practical tasks and comparing the performance of AI tools against traditional methods. The metrics on time and error rates provided comprehensive data on the performance efficiency of each tool. The integration capability was assessed through compatibility tests with different IDEs and operating systems to ensure seamless workflow transitions. The evaluations also included API connectivity and plugin support to gauge how well the tools integrate with external software and data sources. Additionally, the ease of embedding the tools into existing CI/CD pipelines was analyzed to determine the practicality of incorporating the tools into continuous integration and deployment workflows.

Scalability was observed by examining the tools' performance in handling large-scale projects or increased demands, focusing on processing speed and resource utilization. This data collection phase was instrumental in gathering comprehensive and relevant information to evaluate the role of AI tools in accelerating software development. The subsequent sections will build upon this data to interpret findings, compare tools, and assess the impact of prompt engineering.

In summary, this data collection phase was instrumental in gathering comprehensive and relevant information to evaluate AI tools' role in accelerating software development. The subsequent sections will build upon this data to interpret findings, compare tools, and assess the impact of prompt engineering.

5.2 Data Interpretation

The data collection and interpretation phase builds on the data collected in Section 5.1, providing an in-depth analysis of the findings. This section will systematically address each evaluation criterion established in Section 3.2.1, offering a detailed interpretation of the performance, user experience, integration capability, cost-effectiveness, scalability, and security and reliability of the AI tools under study. This structured approach ensures that the insights drawn are directly relevant to the research objectives and practical applications in software development.

5.2.1 Performance Efficiency

The performance efficiency of each AI tool was evaluated based on specific tasks, including implementing tests, developing APIs, altering Swagger documentation, configuring Jenkins pipelines, integrating database connections, and improving existing code.

Speed of Code Generation

The speed of code generation was measured in minutes across various tasks. The data indicates that GitHub Copilot and Custom GPTs consistently performed better in speed, followed closely by ChatGPT 4o and 4.0. For instance, when implementing tests, GitHub Copilot averaged around 7 minutes, significantly improving over the 21.5 minutes taken by manual coding as seen below.

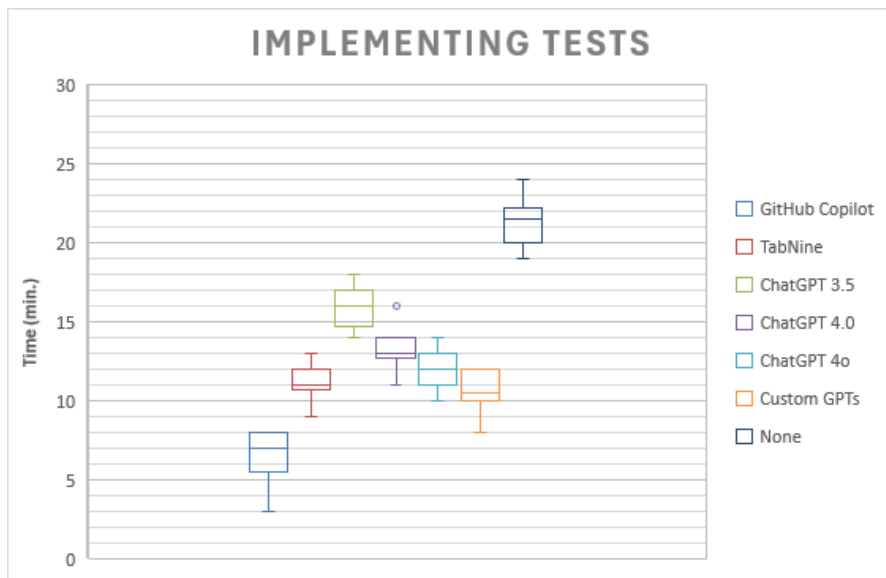


Figure 2 Times for Test Implementing

GitHub Copilot integrated database connections in an average of 18.5 minutes, while Custom GPTs took about 20 minutes. This is significantly faster than the non-AI method, which took around 33 minutes (Figure 3).

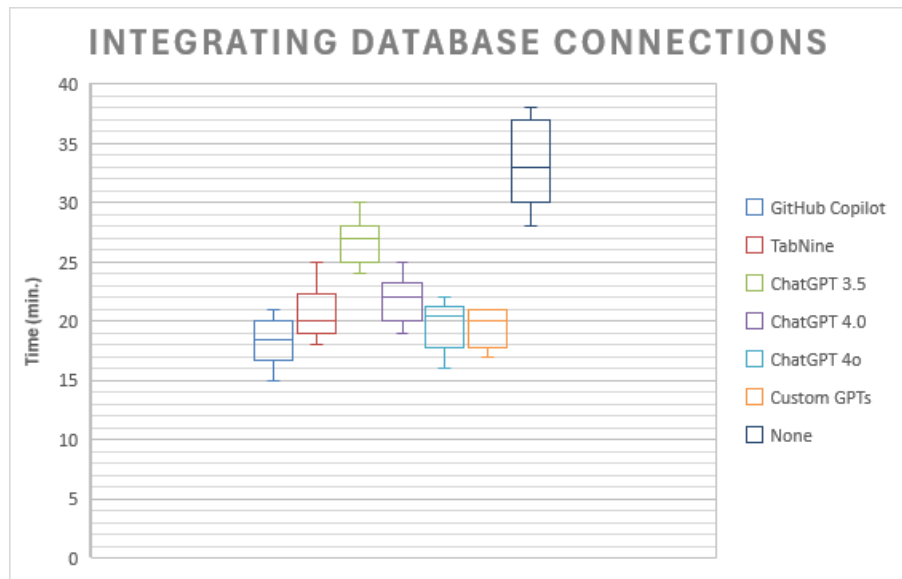


Figure 3 Times for Integrating Database Connections

Similarly, for developing APIs, GitHub Copilot and Custom GPTs averaged around 14 and 16 minutes, respectively, half the time taken by manual coding, which took 29.5 minutes (Figure 4).

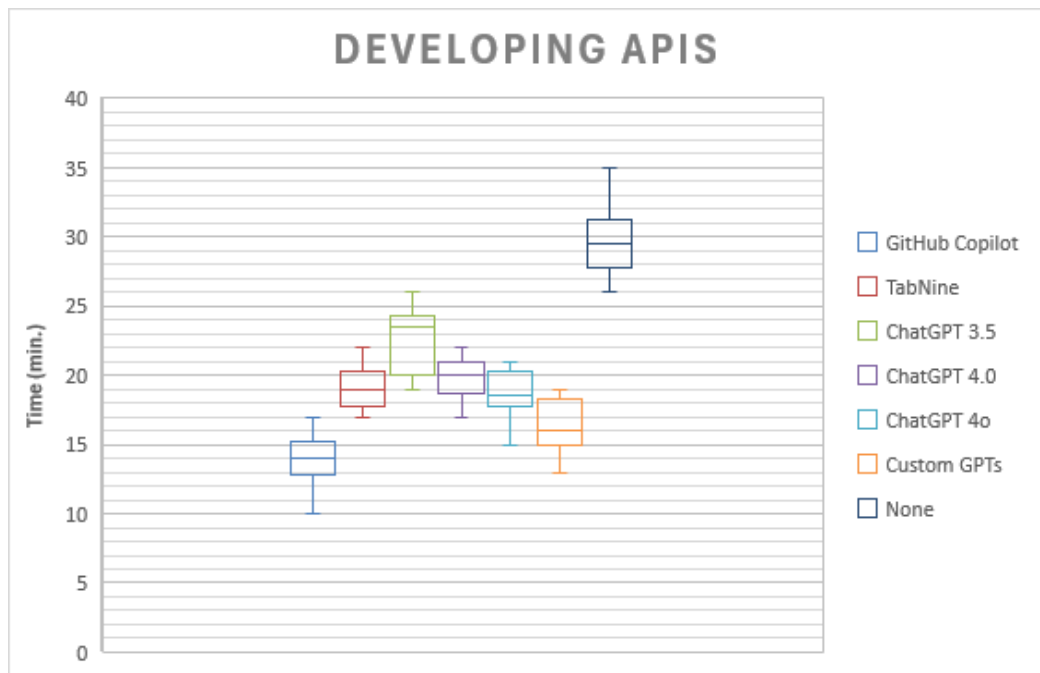


Figure 4 Times for Developing APIs

Altering Swagger documentation saw GitHub Copilot and Custom GPTs completing the task in approximately 4 and 6 minutes, respectively, compared to 9 minutes manually (Figure 5).

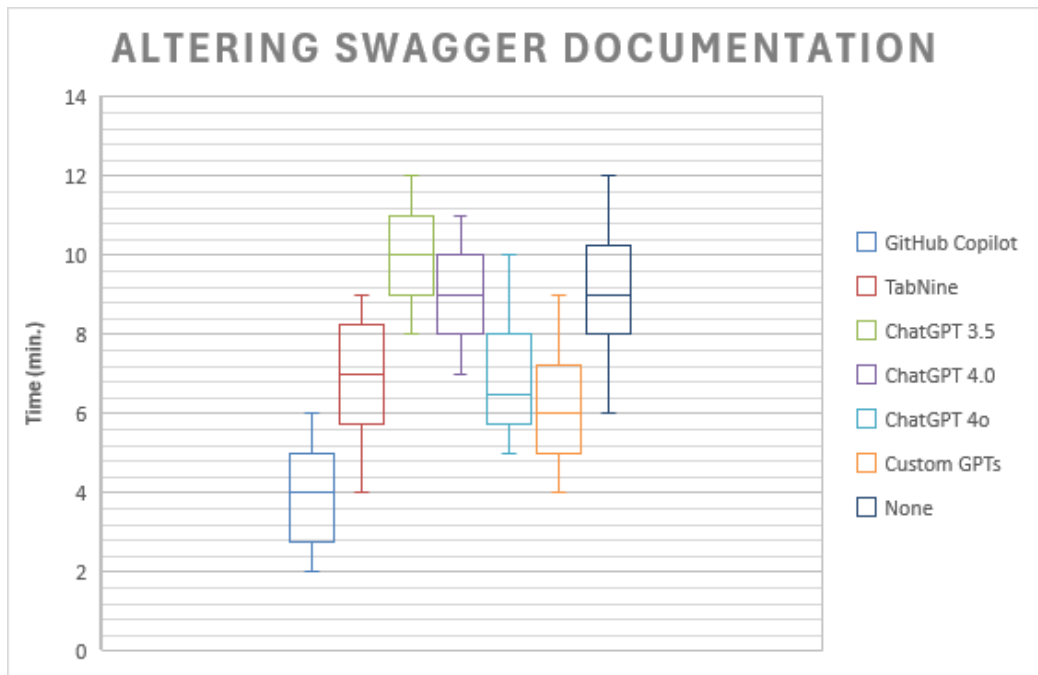


Figure 5 Times for Altering Swagger Documentation

GitHub Copilot averaged around 14 minutes for configuring Jenkins pipelines, demonstrating efficiency over the manual method, which took an average of 20 minutes. It is also essential to see the efficiency of ChatGPT in this task, which showed an improvement of over 50%. Configuring the Jenkins pipeline took only 9.5 and 9 minutes on average for ChatGPT 4.0 and 4o, respectively (Figure 6).

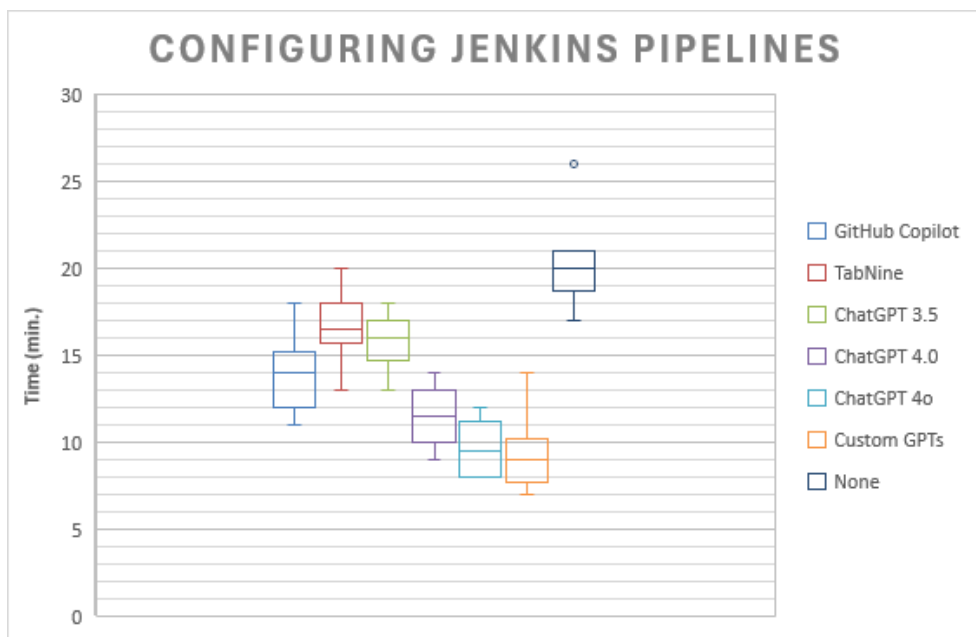


Figure 6 Times for Configuring Jenkins Pipelines

Regarding improving existing code, GitHub Copilot and ChatGPT 4o took about 11 and 6 minutes, significantly faster than the 15.5 minutes by manual methods (Figure 7).

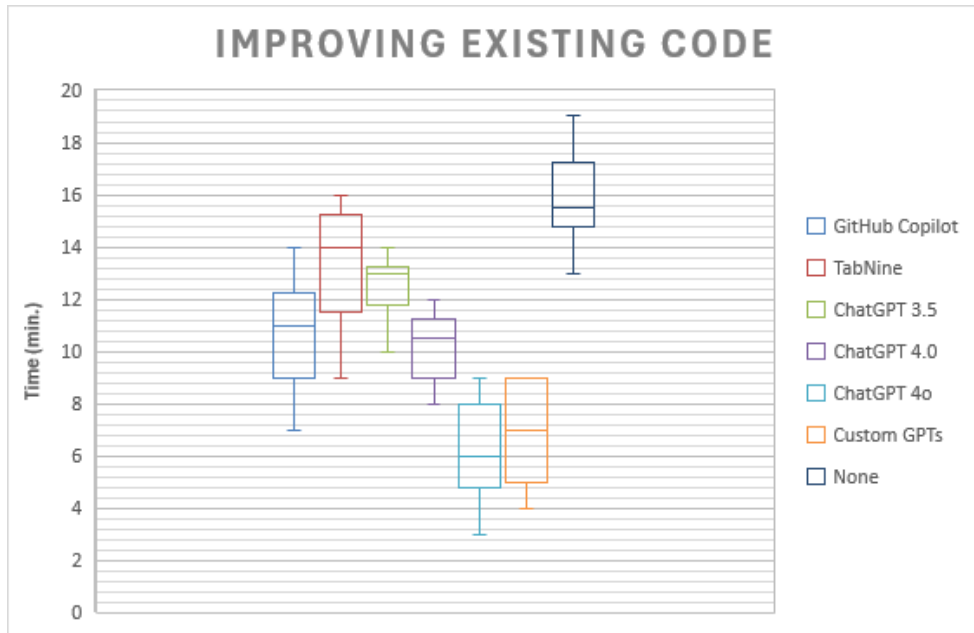


Figure 7 Times for Improving Existing Code

Accuracy of Code Generation

The accuracy of code generation was evaluated by counting the number of errors or issues in the code generated by each tool. The analysis indicated that Custom GPTs and ChatGPT 4o had the lowest number of errors, suggesting higher accuracy in code generation. ChatGPT 4o and Custom GPTs averaged less than one error when implementing tests, whereas non-AI methods had around one to two errors (Figure 8).

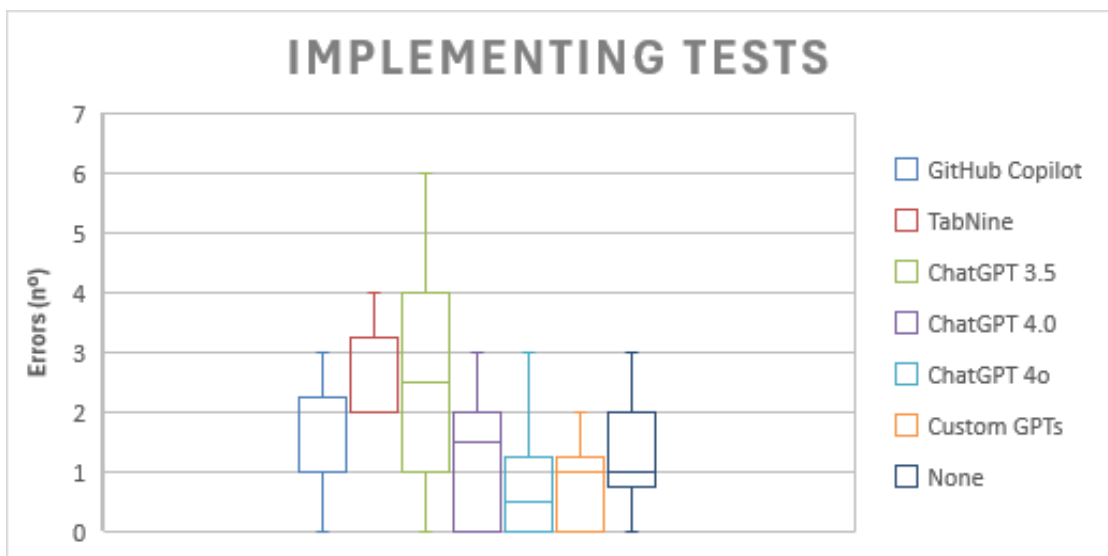


Figure 8 Number of Errors and Issues When Implementing Tests

Integrating database connections was another task where Custom GPTs and ChatGPT 4o had the fewest errors, along with the manual implementation, deviating from zero to around two. In contrast, all other AI methods had, on average, more than 1.5 errors (Figure 9).

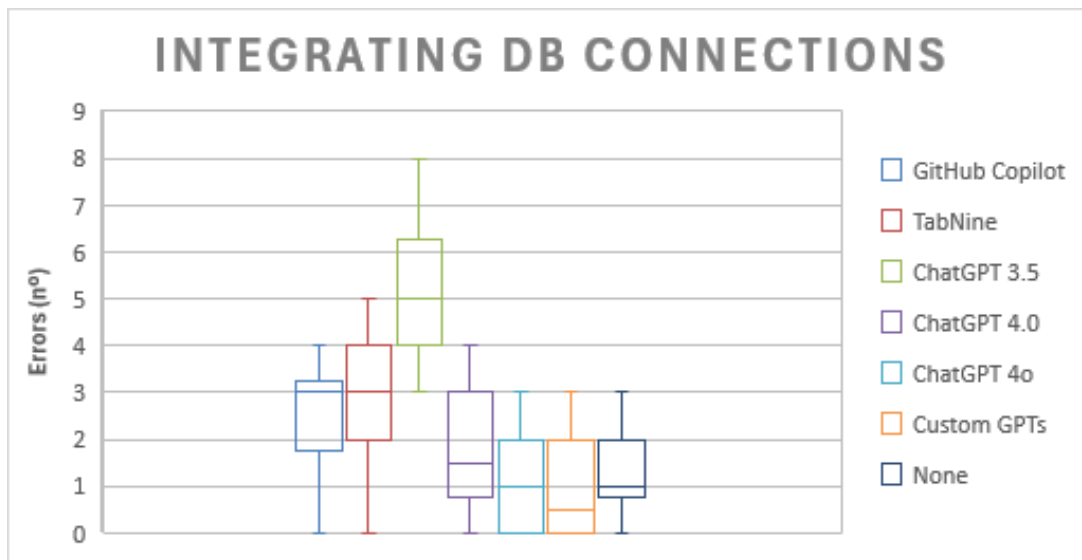


Figure 9 Number of Errors and Issues When Integrating DB Connections

Custom GPTs and ChatGPT 4o again showed superior performance for developing APIs, with Custom GPTs averaging zero errors and ChatGPT 4o around 0.5, compared to one to two errors in every other method (Figure 10).

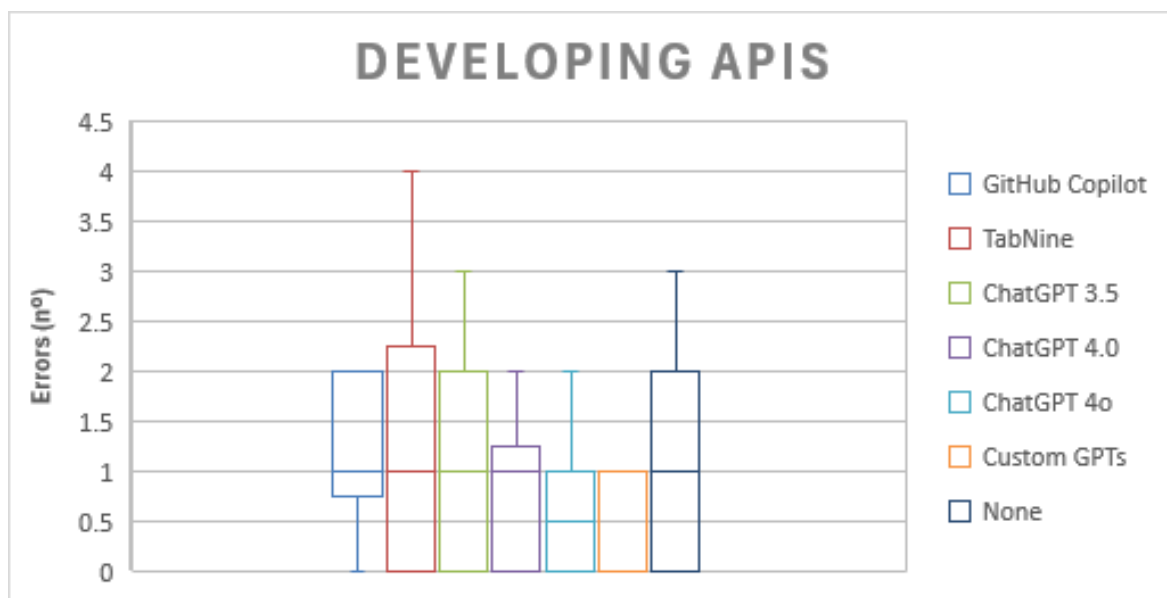


Figure 10 Number of Errors and Issues When Developing APIs

Altering Swagger documentation revealed almost no errors for every method, including the manual alteration of the Swagger documentation; nonetheless, Custom GPTs and ChatGPT 4o could alter the Swagger documentation without any error or issue (Figure 11).

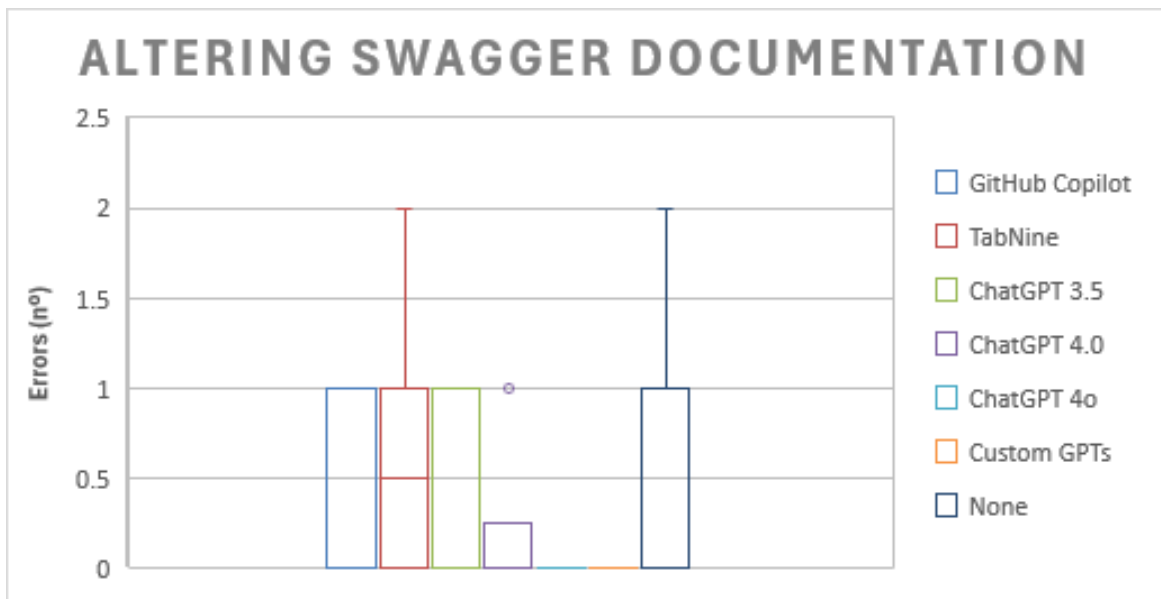


Figure 11 Number of Errors and Issues When Altering Swagger Documentation

In configuring Jenkins pipelines, Custom GPTs demonstrated high accuracy with an average of one error, compared to 3.5 for non-AI methods (Figure 12).

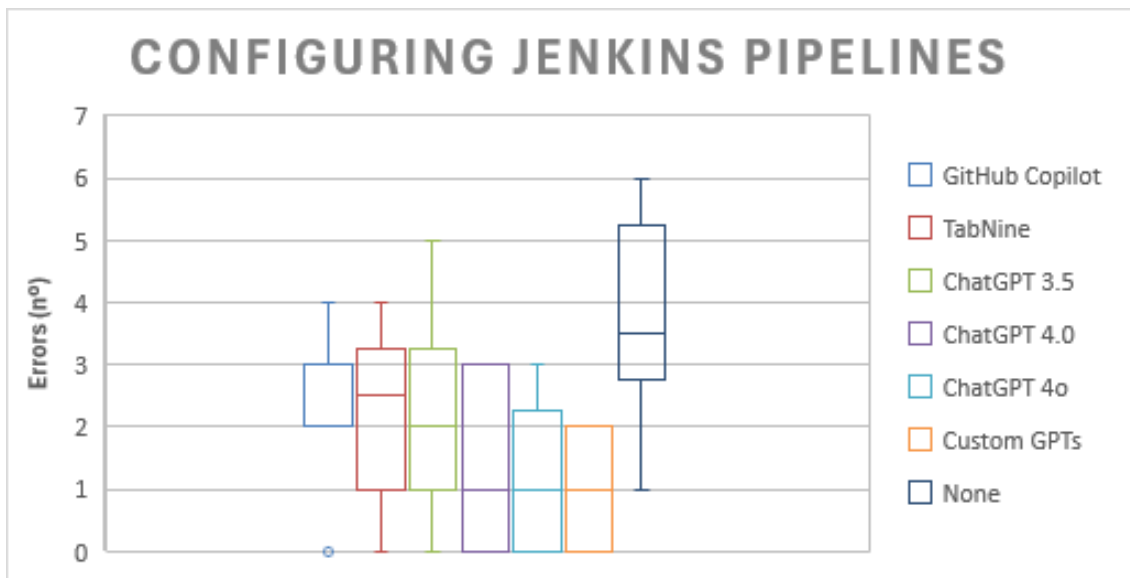


Figure 12 Number of Errors and Issues When Configuring Jenkins Pipelines

For improving existing code, ChatGPT 4o and Custom GPTs showed minimal errors, averaging less than 0.5 errors, even though they had some outliers where the GPT improvement caused some errors, compared to non-AI methods, which averaged about one error (Figure 13).

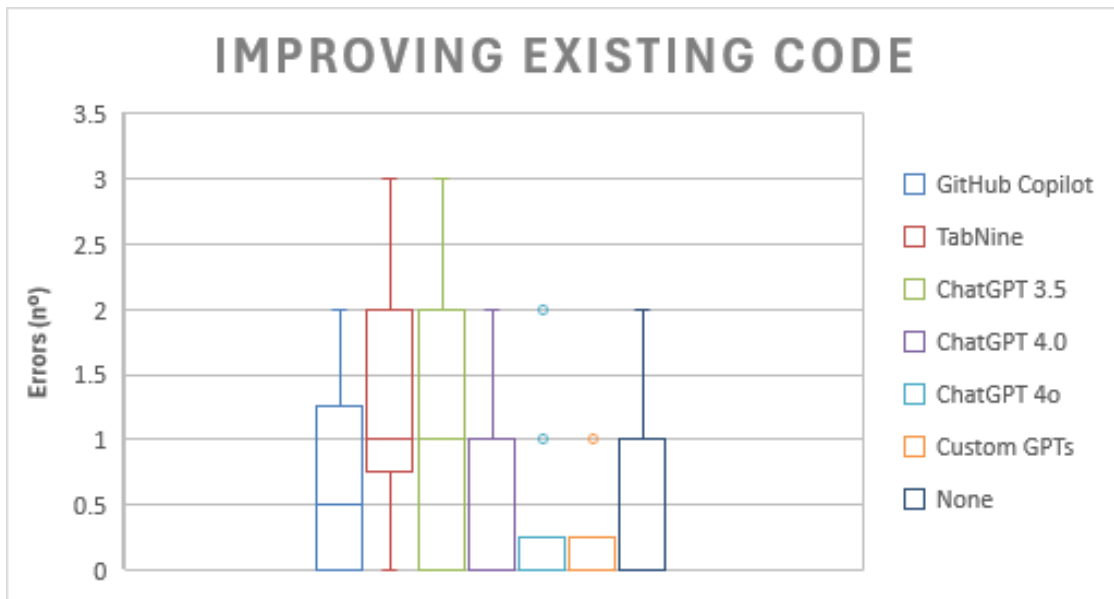


Figure 13 Number of Errors and Issues When Improving Existing Code

The performance efficiency of the AI-driven code generation tools demonstrates notable differences in speed and accuracy. GitHub Copilot emerged as a front-runner, consistently outperforming in speed across various tasks. Its ability to develop APIs, configure Jenkins pipelines, and integrate database connections swiftly underscores its potential for reducing manual coding time significantly. This efficiency is particularly advantageous in rapid development cycles where time constraints are critical.

Custom GPTs also displayed impressive performance, especially in handling complex tasks like improving existing code and altering Swagger documentation. Their high accuracy, evidenced by fewer errors, makes them ideal for projects requiring precision and reliability. Their tailored nature allows them to cater to specific project requirements, enhancing their utility in specialized scenarios.

ChatGPT 4o demonstrated substantial improvements over earlier versions, offering a balanced mix of speed and accuracy. Its performance in tasks such as configuring Jenkins pipelines and developing APIs indicates its robustness and adaptability in various coding environments. While not as fast as GitHub Copilot, ChatGPT 4o's accuracy makes it reliable for maintaining code quality.

TabNine, while effective in providing context-aware suggestions and auto-completion, showed a slight lag in handling more complex scenarios compared to GitHub Copilot and Custom GPTs. Its primary strength is enhancing productivity during the coding phase, although it may require additional manual adjustments for intricate tasks.

In summary, GitHub Copilot and Custom GPTs excel in efficiency, with ChatGPT 4o offering a commendable balance of speed and accuracy. TabNine remains a valuable tool for boosting productivity through its auto-completion features.

5.2.2 User Experience

User experience was assessed through a survey conducted with developers at Celfocus. The survey had eleven questions, some of which were open text, and others were to choose one or more answers. This survey was done before the company purchased the GitHub license, and 279 developers answered.

Familiarity with AI Tools

The survey started with the question “Familiarity with AI-powered Code Assistance Tools.” Of the 279 developers, 89% were familiar with these tools, 43% had only heard of them, and 26% were already using them while working (Figure 14).

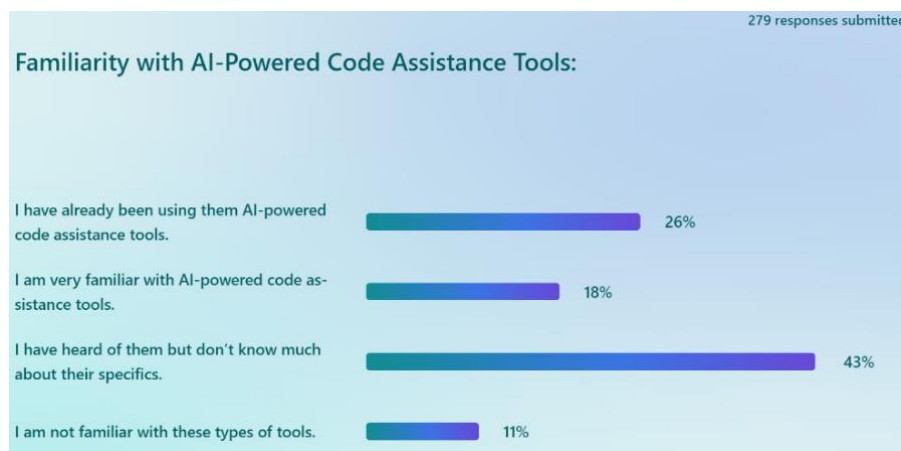


Figure 14 Familiarity with AI-Powered Code Assistance Tools

Following this question, the survey continued with “Which AI-powered code assistance tool have you already used or have been using?” From this question, it was possible to determine that the most popular tool among the developers was ChatGPT, with 60% of usage, followed by GitHub Copilot, with 23% (Figure 15).



Figure 15 AI-Powered Code Assistance Tool Usage

Benefits VS Concerns

To the question, “What potential benefits do you think an AI-powered code assistance tool can bring to our development process?” There were varied answers among the developers. Some of the most pertinent answers were:

- “It will for sure be very positive. The possibility to anticipate any issues, increase performance, and have the best practices implemented.”
- “It can help on certain algorithms that would take longer. Everything can be done. We are just speaking about timesaving. It is like using a calculator; we can do the math on paper, but the calculators are faster. The same works for AI coding.”
- “It will make mundane tasks faster; It will improve how code is written and overall improve everyone's work.”

On the opposite side, when asked, “Do you have any concerns about integrating an AI-powered code assistance tool into our workflow?” the most thrilling concerns were:

- "No, except the security. I think the AI-powered code can be only positive."
- "No, AI will never take away my job. I am not afraid of it. There were some things that I had to teach AI about already. Yes, it can be used as a tool, but it is not always 100% correct. And definitely cannot do the whole job by itself."
- "Relying on AI can also lead to a reduced understanding of the codebase by developers and a false sense of security and AI tools if not properly integrated. They interrupt rather than enhance the flow state."

Productivity, Code Quality, and Security

To evaluate the developers' thoughts on productivity, they were asked, “How do you think an AI-powered code assistance tool will affect your coding productivity?” Of the 279 developers, 90% think it will improve, only 7% think it will not, and none think it would hurt productivity (Figure 16).

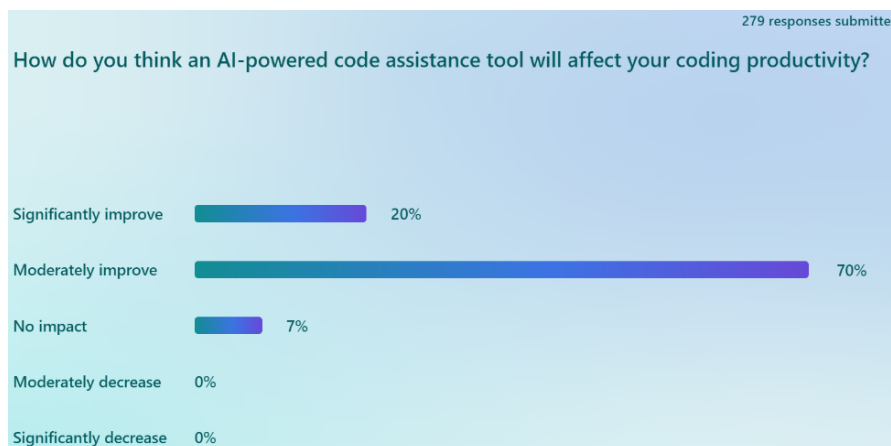


Figure 16 Coding Productivity Answers

For the topic of code quality, the answers to the question, “In your opinion, how will an AI-powered code assistance tool affect the quality of the code we produce?” showed that 81% thought that the code quality would improve, 11% believed that it would not have any impact and only 6% showed some worries about the topic (Figure 17).

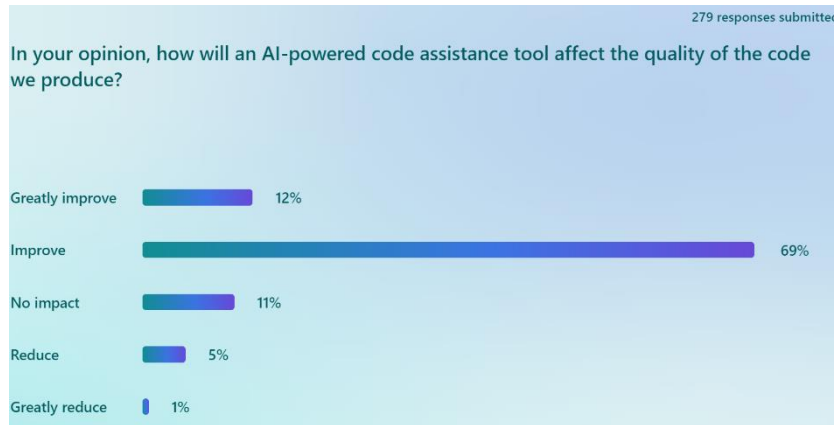


Figure 17 Code Quality Answers

Regarding security, the question, “Are you concerned about security and privacy issues related to using an AI-powered code completion tool?” showed that 65% of the developers are concerned about the security of these tools (Figure 18).

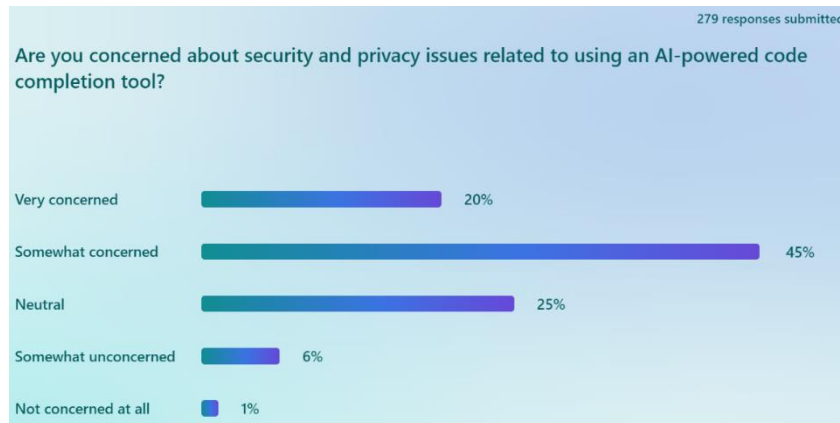


Figure 18 Security and Privacy Answers

Learning Curve

To understand if Celfocus should give the developers licenses for these tools, three questions were asked regarding the learning curve and the willingness to adopt the tools. Starting with “How comfortable are you with the learning curve associated with using such a tool?” The results showed that 79% were comfortable with the learning curve, and only 2% were uncomfortable about the topic (Figure 19).

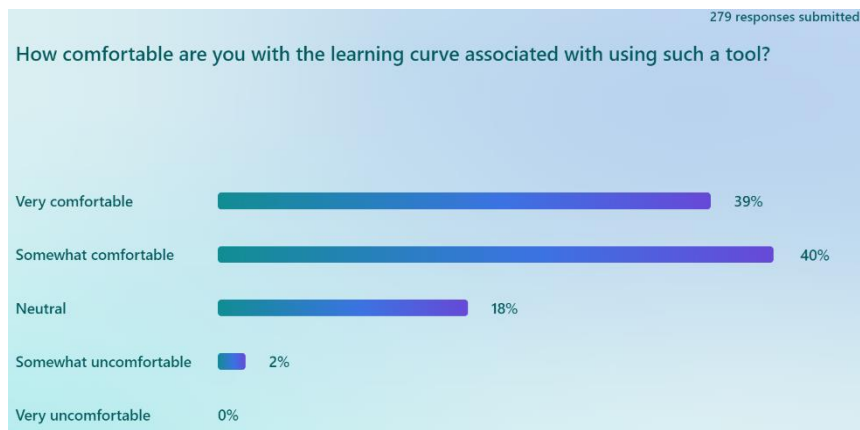


Figure 19 Learning Curve Answers

Following that question, “How willing are you to adopt an AI-powered code assistance tool in your daily work?” showed that 78% were willing to adopt the AI tools, with only 2% showing a lack of interest (Figure 20).

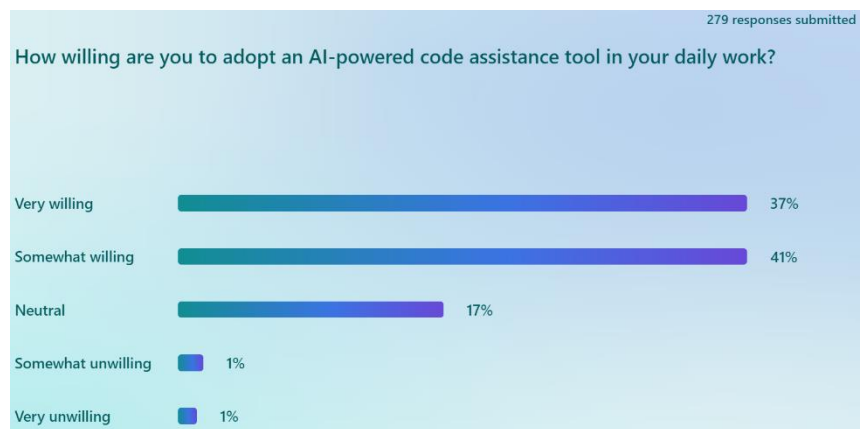


Figure 20 Adoption Willingness Answers

Finally, “What additional training or resources, if any, would you need to effectively use an AI-powered code assistance tool?” showed some exciting answers, of which the most relevant ones were:

- "I couldn't think about additional training, but of course, it would be good to know how to use the tool without compromising the code security."
- "I would need a course related to the tool to understand better its pros and cons and the logic behind every suggestion based on user inputs."
- "Tool-Specific Documentation; Ethical User Guidelines."

The survey closed with the option for the developers to share any additional thoughts or feedback on the potential adoption of an AI-powered code assistance tool. Here, the answers represented a resume of the previous answers, with many not writing anything.

Overall, the survey results indicate a strong interest and positive attitude toward AI-powered code assistance tools among Celfocus developers. While there are notable concerns, particularly around security and potential over-reliance on AI, the perceived benefits in productivity and code quality are compelling. To successfully integrate these tools, Celfocus should address security issues, provide targeted training, and ensure that AI tools complement developers' workflows without disrupting their understanding of the codebase.

5.2.3 Integration Capability

Integration capability was evaluated by examining the tools' compatibility with different IDEs and operating systems, API connectivity, plugin support, and ease of embedding into existing CI/CD pipelines. The findings include:

Compatibility with IDEs and Operating Systems

ChatGPT can be used within IntelliJ and Visual Studio via third-party plugins or extensions. However, it does not have direct integration into these IDEs. It is platform-independent and accessible via Windows, macOS, and Linux web browsers. Using ChatGPT from the web is beneficial because it provides a consistent experience across all operating systems without needing installation or configuration. However, the downside is the lack of seamless integration into IDEs, requiring context-switching between the browser and the development environment.

TabNine is fully compatible with IntelliJ IDEA, Visual Studio, Visual Studio Code, PyCharm, Sublime Text, Atom, and other significant IDEs. It offers cross-platform support for Windows, macOS, and Linux. However, users may experience occasional performance lags in some environments and need to update manually to ensure compatibility with the latest IDE versions.

GitHub Copilot integrates with Visual Studio Code and is developing support for other IDEs, such as IntelliJ IDEA and Visual Studio. It supports Windows, macOS, and Linux through Visual Studio Code. Some users have reported mixed experiences with IntelliJ IDEA and other IDEs, which are still in beta.

Table 2 Compatibility with IDEs and Operating Systems Resume

Tool	IntelliJ & Visual Studio Support	OS	Issues	Web Version Benefits/Drawbacks
ChatGPT	Third-party plugins	Windows, macOS, Linux	Limited direct integration requires additional config	Platform-independent, but requires context switching
TabNine	Full compatibility	Windows, macOS, Linux	Performance lags and manual updates are needed	Not applicable
GitHub Copilot	Developing support	Windows, macOS, Linux	Mixed experiences with non-VS Code IDEs	Not applicable

API Connectivity and Plugin Support

ChatGPT offers robust API connectivity, allowing integration with various applications and workflows. It can pull in data from multiple external sources and depends on third-party plugins for IDEs. Direct plugin support is available for popular platforms like Slack and Microsoft Teams. The web version of ChatGPT is beneficial for quick access to the API without additional setup, but it may not be as tightly integrated as a dedicated IDE plugin.

TabNine primarily focuses on auto-completion within IDEs, with less emphasis on external API connectivity. It has extensive plugin support for various IDEs and works well with other development tools and environments.

GitHub Copilot integrates tightly with GitHub and can pull in data from repositories and other GitHub services, although it is currently limited to the GitHub ecosystem. It has native plugin support in Visual Studio Code and is developing plugins for other significant IDEs.

Table 3 API Connectivity and Plugin Support Resume

Tool	API Connectivity	Plugin Support	Web Version Benefits/Drawbacks
ChatGPT	Robust API integration	Third-party plugins, direct for Slack, Teams	Quick API access, less IDE integration
TabNine	Limited API focus	Extensive IDE plugin support	Not applicable
GitHub Copilot	Tight GitHub integration	Native in VS Code, developing for others	Not applicable

Embedding into CI/CD Pipelines

ChatGPT can be integrated into CI/CD pipelines through API calls for tasks such as generating code snippets, documentation, or code reviews. It is effective for automating documentation and developing test cases, though it requires custom scripts and setup for seamless integration. The web version is beneficial for quick setup and testing of API calls but may not be as streamlined as a direct IDE or CI/CD tool integration.

TabNine is not typically embedded into CI/CD pipelines but enhances the development phase before committing code. It improves developer productivity during the coding phase but has limited direct impact on CI/CD processes.

GitHub Copilot can generate code and automate repetitive tasks before committing to the CI/CD pipeline. It can potentially assist in developing scripts and configuration files for the pipeline, speeding up development and reducing errors before code enters the CI/CD pipeline. However, it is still in the initial stages of direct CI/CD pipeline integration.

Table 4 Embedding into CI/CD Pipelines Resume

Tool	CI/CD Integration	Results	Web Version Benefits/Drawbacks
ChatGPT	API calls, custom scripts	Automates documentation, test case generation	Quick testing setup, not streamlined for CI/CD
TabNine	Enhances development phase	Improves productivity, limited CI/CD impact	Not applicable
GitHub Copilot	Pre-commit automation	Speeds up development, early CI/CD integration	Not applicable

The analysis revealed distinct advantages and limitations for ChatGPT, TabNine, and GitHub Copilot across critical areas of compatibility, API connectivity, plugin support, and CI/CD pipeline integration.

ChatGPT is highly versatile, offering platform-independent access through web browsers on Windows, macOS, and Linux. Its web-based nature allows quick and easy access to its robust API, making it useful for generating code snippets, documentation, and performing code reviews. However, the lack of seamless integration into popular IDEs like IntelliJ and Visual Studio necessitates context-switching and reliance on third-party plugins. While this flexibility is valuable, it may impede a fully streamlined development workflow.

TabNine provides comprehensive autocompletion support across various IDEs, including IntelliJ IDEA, Visual Studio, and Visual Studio Code. Its cross-platform support ensures consistency across different operating systems, though occasional performance issues and the need for manual updates can be drawbacks. TabNine's strength lies in enhancing developer

productivity during the coding phase, but its impact on CI/CD processes is limited, as it is not typically embedded into these pipelines.

GitHub Copilot offers tight integration with GitHub services and native support for Visual Studio Code, with ongoing development for other major IDEs. This integration facilitates the automation of repetitive coding tasks and script generation, potentially reducing errors and accelerating development before code enters CI/CD pipelines. However, its full potential in CI/CD integration is still evolving, and experiences with IDEs outside of Visual Studio Code have been mixed.

In conclusion, each tool has unique benefits: ChatGPT is highly adaptable with powerful API capabilities, TabNine offers extensive IDE support and productivity enhancements, and GitHub Copilot provides deep integration with GitHub and automation potential. The choice between these tools will depend on specific workflow requirements, the importance of seamless IDE integration, and the need for automation within CI/CD pipelines.

5.2.4 Cost-effectiveness

A detailed cost-benefit analysis was performed to assess each tool's cost-effectiveness. This included direct costs such as purchase and operational expenses and indirect costs like user training and downtime during implementation.

Direct Costs

Direct costs encompass the purchase and operational expenses associated with each tool. These are straightforward costs incurred due to using the tool and are easier to quantify. The table below summarizes each tool's monthly, daily, and hourly costs, including subscription fees and any additional costs required.

Table 5 Tools Direct Cost

Tool	Cost Per Month (\$)	Cost Per Day (\$)	Cost Per Hour (\$)
ChatGPT 4.0 & ChatGPT 4.0 & Custom GPTs	60.00	2.73	0.34
TabNine	39.00	1.77	0.22
Copilot	19.00	0.86	0.11

The Table 5 shows that Copilot has the lowest direct costs, making it an attractive option for cost-conscious users. TabNine is moderately priced, while ChatGPT 4.0, ChatGPT 4.0, and Custom GPTs are slightly higher.

Indirect Costs

Indirect costs include expenses that are not directly tied to the purchase or subscription of the tool but are incurred during its usage. These can include training costs, implementation

downtime, and additional time or resources needed to integrate the tool into existing workflows. The table below details the training hours required for each tool, the hourly rate for a developer (in Celfocus, a developer costs \$250 per day), and the total training cost.

Table 6 Indirect Costs

Training Hours (All AI Tools)	Rate Per Hour (\$)	Total Cost (\$)
5 Hours	31.25	156.25

Training costs are calculated based on the time required to get developers up to speed with the new tools. Each tool requires approximately five hours of training, which, at an hourly rate of \$31.25, totals \$156.25. This cost is a one-time investment and can vary depending on the developer’s familiarity with similar tools and the complexity of the tool itself.

Cost-Benefit Analysis

The cost-benefit analysis compares the overall costs (both direct and indirect) against the benefits provided by each tool. Benefits are measured in terms of reduced coding time, decreased need for debugging, and overall efficiency improvements. The net savings for different hours saved using each tool are shown below. Positive values indicate savings, while negative values represent costs.

Table 7 Cost-Benefit Analysis

Time Saved (hours)	ChatGPT 4.0 & ChatGPT 4.0 & Custom GPTs (\$)	TabNine (\$)	Copilot (\$)
0.00	-0.34	-0.22	-0.11
0.10	3.92	4.04	4.15
0.20	8.18	8.30	8.41
0.30	12.44	12.56	12.68
0.40	16.70	16.82	16.94
0.50	20.97	21.09	21.20
0.60	25.23	25.35	25.46
0.70	29.49	29.61	29.72
0.80	33.75	33.87	33.98
0.90	38.01	38.13	38.24
1.00	42.27	42.39	42.51

The Table 7 shows the net savings achieved using the tools to save time. The first column shows the amount of time saved in hours. The other columns represent the amount of money it costs Celfocus, the negative values, and the amount of money Celfocus makes in profit from each tool, considering the price of each tool.

The Figure 21 helps Celfocus better understand the cost dynamics and potential savings by visually representing the money saved without considering indirect costs for each tool, same values as the table but in form of a graph.

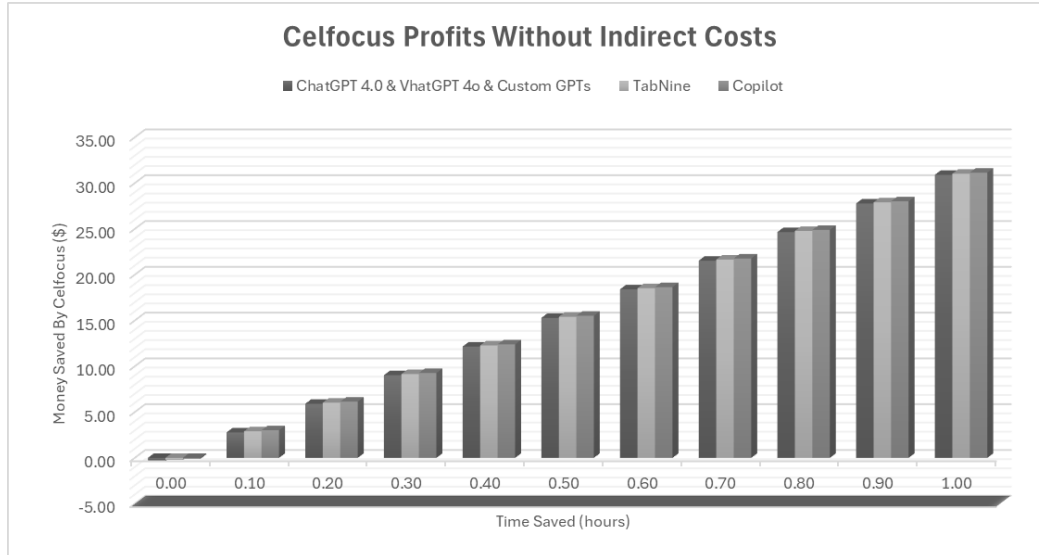


Figure 21 Celfocus Profits Without Indirect Costs

The analysis indicated that each tool's cost-effectiveness varies significantly when considering direct and indirect costs. Direct costs are more straightforward, with Copilot emerging as the most cost-effective option at \$19.00 per month, followed by TabNine at \$39.00 per month, and ChatGPT 4.0 & Custom GPTs at \$60.00 per month. The substantial cost of a developer at Celfocus underscores the potential value of these tools in reducing overall expenses.

Indirect costs, primarily associated with training, were calculated to be a one-time investment of \$156.25 per developer. This highlights the importance of the tools' efficiency in reducing development time to offset these initial costs.

The cost-benefit analysis demonstrated that each tool provides significant net savings with increased usage time. For instance, saving one hour of development time yields a net saving of approximately \$42 across all tools.

The graphical representation further supports these findings by illustrating the incremental savings as time saved increases. This visual aid underscores each tool's potential return on investment. It is also essential to understand that each tool needs to save a developer at least five hours to recover the initial training costs, making the training investment “free.”

In conclusion, considering the performance results of each tool, GitHub Copilot is the most cost-effective tool, given its high-performance efficiency, low direct cost (\$19.00/month), and significant savings per hour of development time. The cost-benefit analysis shows that its ability to rapidly reduce manual coding time translates directly into higher net savings.

Although TabNine is effective, its higher monthly cost (\$39.00/month) and slightly lower performance efficiency result in lower overall savings than GitHub Copilot. It remains a strong contender but is less cost-effective.

Finally, ChatGPT has the highest monthly cost (\$60.00/month) but excels in accuracy and handling complex tasks. Its cost-effectiveness is realized in projects requiring high precision, where its superior accuracy justifies the higher expense. The cost-benefit analysis indicates that it provides significant savings but is less suitable for projects where speed and low cost are primary concerns.

In summary, GitHub Copilot is the most cost-effective tool due to its low-cost and high-performance efficiency balance. TabNine offers robust performance but at a higher cost, making it less economical. ChatGPT is best suited for projects demanding high precision despite its higher price. Each tool's suitability depends on the development project's specific requirements and budget constraints.

5.2.5 Scalability

Scalability was assessed by observing the tools' performance in handling large-scale projects and increasing demands. Factors such as processing speed and system resource utilization were considered.

Handling Large-Scale Projects

ChatGPT 3.5 is a versatile tool that can handle various coding tasks, but it has limitations when dealing with large-scale projects. Its ability to maintain context across extensive codebases is moderate, often leading to inconsistencies that require manual correction. While ChatGPT 3.5 can follow project requirements and generate relevant code snippets, it sometimes struggles with complex architectural suggestions. Significant user intervention is needed to ensure coherence and alignment with project goals. This makes it less suitable for substantial projects where maintaining continuity and overarching design is crucial (Mudgal and Wouhaybi, 2023).

ChatGPT 4.0 represents a significant improvement over its predecessor, particularly in handling large-scale projects. It excels in understanding complex project requirements and maintaining context across extensive codebases. This version can effectively suggest architectural improvements and optimize code structure, making it highly suitable for large projects. Minimal user intervention is needed as ChatGPT 4.0 can maintain high coherence and consistency throughout the project, facilitating smoother development processes (Dong et al., 2024).

As ChatGPT 4o offers better performance than ChatGPT 4.0, it is to expect a display of similar effectiveness in managing large-scale projects. But as it is a new model there is still no data on the matter.

Custom GPTs offer a tailored to specific project requirements version of ChatGPT 3.5. It is expected again that it will have remarkable capabilities in handling large-scale projects. As studies have not been made for specific Custom GPTs there is still no data that proves it.

TabNine is a highly efficient tool for generating code snippets and providing context-aware suggestions. It integrates well with development environments, making it a valuable tool for developers working on large projects. However, more human oversight is required to ensure coherence and alignment with project goals in large-scale projects. While effective in providing immediate coding assistance, it lacks the deeper project understanding and architectural suggestion capabilities in more advanced GPT models (Sun et al., 2023).

GitHub Copilot excels in providing context-aware code suggestions and integrates seamlessly with various development environments. It is highly effective in generating relevant code snippets but requires significant human oversight to ensure coherence in large-scale projects. Like TabNine, it cannot provide comprehensive architectural suggestions, making it more suitable for assisting with specific coding tasks rather than overall project management (Sun et al., 2023).

Table 8 Handling Large-Scale Projects Resume

Tool	Context Management	Code Coherence	Architectural Suggestions	User Intervention
ChatGPT 3.5	Struggles with substantial projects	Can be inconsistent	Limited ability	Significant oversight needed
ChatGPT 4.0	Excels in maintaining context	High consistency	Strong ability	Minimal oversight needed
ChatGPT 4o	Expected to be better than 4.0	N/A	N/A	N/A
Custom GPTs	Expected to be better or on par with 4.0	N/A	N/A	N/A
TabNine	Effective for context-aware suggestions	Requires human oversight	Limited ability	Significant oversight needed
GitHub Copilot	Excellent for context-aware suggestions	Needs human oversight	Limited ability	Significant oversight needed

System Resource Utilization

All versions of ChatGPT were tested using a browser, so the memory management and CPU utilization were not considered, as the memory and CPU usage were from the browser and did not change with the usage of ChatGPT.

ChatGPT 3.5 offers moderate processing speed. Its model is the least advanced of the versions resulting in longer response times than its successors (Mamgain, 2024).

ChatGPT 4.0 demonstrates fast processing speeds. Its improved model ensures quick response times even during intensive tasks. This optimization makes it a robust tool for large projects enhancing productivity and reducing downtime (Mamgain, 2024).

Like ChatGPT 4.0, ChatGPT 4o exhibits the fastest processing speeds due to its newer model, resulting in quick response times (Mamgain, 2024).

Custom GPTs demonstrate speeds like 4.0 as it uses the ChatGPT 3.5 model but optimized for specific tasks (Mamgain, 2024).

TabNine offers fast processing speeds, but its system resource utilization can increase during intensive tasks. It consumes considerable memory and CPU resources, especially when processing large volumes of code or complex operations. Despite this, it maintains a smooth integration with development environments, ensuring developers can work efficiently without significant delays (Yahav, 2021).

GitHub Copilot efficiently utilizes resources, offering fast processing speeds. However, it can become resource-intensive during intensive tasks, consuming significant memory and CPU resources. Despite this, it provides quick response times and integrates well with various IDEs, ensuring a smooth user experience even under demanding conditions (Cerutti, 2023).

Table 9 System Resource Utilization Resume

Tool	Processing Speed	Memory Management	CPU Utilization	Response Time
ChatGPT 3.5	Moderate	N/A	N/A	Adequate
ChatGPT 4.0	Fast	N/A	N/A	Quick
ChatGPT 4o	Very Fast	N/A	N/A	Very Quick
Custom GPTs	Fast	N/A	N/A	Quick
TabNine	Fast	High but resource-intensive	High resource consumption	Generally quick, may slow under load
GitHub Copilot	Fast	High resource consumption	High resource consumption	Generally quick, may slow under load

The analysis revealed that each tool has strengths and limitations in handling large-scale projects and utilizing system resources. ChatGPT 4.0 stands out for its superior performance in maintaining context and coherence across extensive codebases coupled with its high processing speeds, and low response times, making it highly suitable for complex and demanding projects. ChatGPT 3.5, while versatile, shows limitations in processing speeds and response times.

TabNine and GitHub Copilot excel at generating context-aware code suggestions and seamlessly integrate with development environments. However, they require significant human oversight to ensure coherence in large-scale projects and tend to be more resource-intensive during intensive tasks.

Custom GPTs, with their tailored optimization for specific project requirements, exhibit remarkable project handling and resource efficiency capabilities, making them the ideal choice for large and complex projects. In contrast, ChatGPT 4o offers performance comparable to ChatGPT 4.0, demonstrating effective project management, high processing speeds, and low response times.

Overall, the choice of tool depends on the project's specific needs, with ChatGPT 4.0 and Custom GPTs being the most balanced in handling large-scale projects and speed.

5.2.6 Security and Reliability

Stress tests were conducted to evaluate the robustness of each tool under various conditions. Historical data on known security issues or downtime incidents were also reviewed. Study on the security improvements from the enterprise licenses. The findings include:

Stress Tests

Under stress testing conditions, ChatGPT 3.5 was subjected to high volumes of concurrent user requests and prolonged usage periods to evaluate its performance stability. The tool demonstrated impressive resilience, maintaining response times within acceptable ranges and without significant degradation in user experience. The tests revealed that the system's underlying infrastructure could scale efficiently to handle the increased load, although occasional latency spikes were observed during peak usage times (Curry, 2024).

During stress testing, ChatGPT 4.0 was evaluated for its performance under extreme conditions, including high user concurrency and extended operational periods. The tool exhibited excellent performance and minimal impact on response time and accuracy, even under maximum load. This resilience is attributed to its enhanced infrastructure, which supports dynamic scaling and load balancing, ensuring consistent user experience regardless of the load (Curry, 2024). ChatGPT 4o was revealed recently, so it has not received many tests. Nonetheless, its capability to handle high levels of user activity and data processing without performance degradation is a must because millions of users use GPT 4.0 and migrate to 4o.

The performance of Custom GPTs under stress testing varied based on their individual configurations and deployment environments. These tools performed well, maintaining satisfactory response times and stability under high loads. However, the bespoke nature of Custom GPTs meant that some instances required additional optimization to achieve consistent performance under stress.

TabNine's performance under stress testing conditions has been exemplary. The tool efficiently handled high volumes of simultaneous requests and prolonged operational periods without significant performance drops. This capability is attributed to its optimized algorithms and scalable architecture, ensuring seamless integration with various development environments even under heavy use.

GitHub Copilot performed exceptionally well under stress testing, maintaining high performance levels even with many concurrent users and extensive usage scenarios. The tool’s scalable infrastructure and efficient resource management contributed to its ability to handle stress without significant performance impacts.

Table 10 Stress Tests Resume

Tool	Stress Tests
ChatGPT 3.5	High resilience, minor latency spikes during peak usage
ChatGPT 4.0	Excellent performance, minimal impact on response time and accuracy
ChatGPT 4o	N/A
Custom GPTs	N/A
TabNine	Exemplary performance, efficient handling of high volumes and prolonged periods
GitHub Copilot	Exceptional performance, scalable infrastructure, efficient resource management

Historical Security Issues

Since its release, no significant security vulnerabilities have been reported for ChatGPT 3.5. The development team has proactively addressed minor issues through regular updates and patches. Downtime incidents have been rare and typically related to scheduled maintenance, with transparent communication to users. The tool’s historical reliability record contributes to its strong reputation in the market (Qammar et al., 2023).

ChatGPT 4.0 has maintained a clean security record with no reported vulnerabilities or breaches. The proactive approach to security, including regular updates and a responsive support team, has minimized potential risks. Historical downtime has been minimal, primarily associated with planned updates and infrastructure enhancements, with users kept informed throughout the process (Qammar et al., 2023).

Since ChatGPT 4o is based on 4.0, no significant security incidents are expected to be reported. Again, since custom GPTs are based on GPT 3.5, they will have similar results.

No major security incidents have been reported for TabNine. The tool's proactive approach to security, including regular updates and a dedicated security team, has ensured its resilience against potential threats (“Tabnine | Total code privacy & zero data retention,” n.d.).

GitHub Copilot has maintained a strong security track record and reported no significant vulnerabilities or breaches. One of the known issues with the Copilot Individual license is that it can replicate vulnerabilities. If vulnerabilities already exist in the current project, Copilot may suggest code that contains those vulnerabilities (Degges, 2024). Downtime incidents have been infrequent, primarily related to scheduled maintenance, ensuring high availability and reliability for users.

Table 11 Historical Security Issues Resume

Tool	Historical Security Issues
ChatGPT	No significant vulnerabilities, minor issues addressed, rare downtime
ChatGPT 4.0	Clean security record, minimally planned downtime
ChatGPT 4o	N/A
Custom GPTs	N/A
TabNine	No significant incidents, proactive updates, rare downtime
GitHub Copilot	Strong security track record, can replicate vulnerabilities in the individual license, infrequent downtime

Enterprise licenses

Enterprise licenses for AI-powered code assistance tools offer enhanced security and privacy features, which are crucial for organizations handling sensitive code and data. These licenses ensure that the code remains private and is not used to train AI models, thus safeguarding intellectual property and sensitive information.

ChatGPT's enterprise version provides robust security measures that prevent code and user data from being used to train the model. This feature ensures complete privacy and compliance with corporate data protection policies. The enterprise version includes advanced threat detection and state-of-the-art encryption, further enhancing its security framework. These measures ensure that sensitive code and data remain confidential and secure, providing peace of mind to enterprise users (OpenAI, n.d.).

GitHub Copilot's enterprise license offers additional security features, including automatic removal of any information that may contain security vulnerabilities ("Configuring content exclusions for GitHub Copilot," 2024). This proactive approach helps prevent the propagation of insecure code suggestions, which is crucial in maintaining code integrity. Moreover, the enterprise version benefits from GitHub's robust infrastructure, ensuring high availability and reliability while maintaining stringent security standards.

TabNine's enterprise version also prioritizes security and privacy, ensuring that code is not used to train the AI models. This version includes comprehensive privacy controls and advanced security features like zero data retention policies and encrypted data handling. These measures are designed to protect sensitive information and maintain the confidentiality of the code, making it a secure choice for enterprises ("Plans & Pricing | Tabnine," n.d.).

The analysis revealed that all the tested tools—ChatGPT 3.5, ChatGPT 4.0, ChatGPT 4o, Custom GPTs, TabNine, and GitHub Copilot—demonstrate a high level of security and performance under stress conditions, with each tool excelling in different aspects. The enterprise versions of these tools offer additional security benefits, ensuring that code remains private and secure.

ChatGPT 4.0 and TabNine displayed advanced threat detection mechanisms and state-of-the-art encryption, while GitHub Copilot's enterprise version mitigates security risks by removing

sensitive information from code suggestions. These robust security frameworks and proactive approaches to security updates and patches significantly minimize potential risks.

Overall, these tools are well-equipped to handle enterprise environments' security and performance demands. Their robust security measures, solid performance under stress, and commitment to maintaining code privacy make them reliable choices for organizations seeking AI-powered assistance in development.

5.2.7 Summary of Findings

The data interpretation phase reveals that AI-driven code generation tools have significant potential to transform software development practices. GitHub Copilot and Custom GPTs, particularly ChatGPT 4.0 and 4o, stand out for their performance efficiency, integration capabilities, and scalability. TabNine offers valuable support for enhancing productivity, although it may require additional oversight for complex tasks.

The findings underscore the importance of prompt engineering in maximizing the utility of these tools and highlight best practices for effective, prompt design.

The integration of AI in programming offers promising opportunities for accelerating development processes and improving code quality. However, it also presents challenges that must be carefully navigated, such as ensuring seamless integration, maintaining security and reliability, and optimizing prompt engineering techniques. By addressing these challenges, the potential of AI-driven code generation tools can be fully realized, driving innovation and efficiency in software development.

5.3 Prompt Effectiveness and Construction

This section examines the role of prompt engineering in the effectiveness of AI-driven code generation tools. By analyzing the construction and results of various prompts, we can identify patterns and best practices that enhance these tools' performance. All examples will be made using GPT 4o.

5.3.1 Importance of Prompt Engineering

Prompt engineering is crucial for maximizing the utility of AI tools like GitHub Copilot, TabNine, and ChatGPT. The quality of prompts directly influences the generated code's accuracy, relevance, and efficiency. Effective prompts can lead to significant improvements in code quality and developer productivity. Clear and specific prompts tend to yield more accurate and relevant code suggestions. Providing sufficient context within the prompt helps the AI tool understand the intended functionality. Balancing brevity and necessary detail is essential. Too much detail can overwhelm the AI, while too little can lead to vague or incorrect outputs.

5.3.2 Analysis of Prompt Effectiveness

The analysis focuses on various prompts used during the experiments with the AI tools. Specific criteria were used to evaluate each prompt's construction and the resulting code's effectiveness.

Examples of Effective Prompts

- Test Implementation

[Annex 3 - Effective Prompt for Test Implementation

]

- API Development

[Annex 4 - Effective Prompt for API Development

]

Examples of Ineffective Prompts

- Vague Instructions

[Annex 5 - Ineffective Prompt with Vague Instructions

]

- Lack of Context

Generate code for the project



Sure, I can help with that! To provide you with the most relevant code, I need more details about the project you are working on. Here are a few questions that can help narrow it down:

1. **What is the project's objective?** (e.g., a web application, a mobile app, a data analysis script)
2. **What technologies or programming languages are you using?** (e.g., Python, JavaScript, Java, etc.)
3. **Any specific frameworks or libraries?** (e.g., Django, React, Flask)
4. **Key functionalities or features you need in the project.**
5. **Any specific requirements or constraints?**

Once I have more details, I can generate code snippets or even a complete project skeleton for you.

Figure 22 Lack of Context Result

5.3.3 Impact of Prompt Design on Results

The design of prompts significantly impacts the AI tool's performance. Detailed analysis of how different prompt designs affected the outcomes is provided below:

Detailed Prompts

Detailed prompts often resulted in high-quality, functional code that required minimal adjustments.

[Annex 6 - Detailed Prompt

]

Concise Prompts

Concise prompts were effective when the task was straightforward but led to subpar results for complex tasks.

[Annex 7 - Concise Prompt

]

Contextual Prompts

Providing contextual information within the prompt improved the relevance and accuracy of the code.

[Annex 8 - Contextual Prompt

]

5.3.4 Best Practices for Prompt Engineering

Based on the analysis, several best practices for prompt engineering have been identified to optimize the effectiveness of AI tools. The user needs to be specific and clear, include context, use step-by-step instructions, and iteratively refine prompts based on initial outputs to improve the quality of subsequent code generation.

Creating a "God Prompt" for prompt generation was possible using the ChatGPT.

☆
God Prompt

Please forget all prior prompts. I want you to become my Prompt Creator. Your goal is to help me build the best detailed prompt for my needs. This prompt will be used by you, ChatGPT. Please follow this following process:

<p>1. Your first response will be ask me what prompt should be about.</p> <p>I will provide my answer, but we need to improve it through continual iterations by going through the next steps.</p>	<p>2. Based on my input, you will generate - 3 sections.</p> <p>a) Revised prompt [provide your rewritten prompt. It should be clear, concise, and easily understood by you].</p> <p>b) Suggestions [provide suggestions on what details to include in the prompt to improve it] and</p> <p>c) Questions [ask any relevant questions pertaining to what additional information is needed from me to improve the prompt].</p>	<p>3. We will continue this iterative process with the providing prompt section until it's complete.</p> <p>If you understand, simply respond with "Read".</p>
---	---	---

Figure 23 "God Prompt" for Prompt Generation

By introducing the prompt in the Figure 23 it was possible to create a GPT to generate prompts using all the best practices for prompt engineering. The prompts generated by this GPT were always specific, included context, and used step-by-step instructions. The user could constantly iteratively refine the prompt based on the initial output if needed.

5.3.5 Summary of Findings

The analysis of prompt effectiveness and construction has highlighted the critical role of prompt engineering in leveraging AI tools for code generation. Effective prompts significantly enhance the generated code's accuracy, relevance, and efficiency, leading to better integration and reduced manual adjustments.

Summary of Key Findings:

- Specific, clear, and contextual prompts yield the best results.
- Detailed prompts are more effective for complex tasks.
- Iterative refinement of prompts can continuously improve code quality.

Adhering to these best practices can help developers optimize AI-driven code generation tools, enhancing productivity and code quality in software development projects.

6 Discussion

This chapter explores the implications of integrating AI-driven tools in programming and code generation. It discusses the practical applications and potential benefits of using these tools in real-world software development scenarios, highlighting how they can enhance productivity, accuracy, and efficiency. Additionally, it addresses the study's limitations, acknowledging the constraints and areas that require further research. The discussion aims to provide a balanced view of AI's role in modern programming, considering both the advantages and challenges of incorporating such advanced technologies into existing workflows.

6.1 Integration of AI in Programming

The integration of AI in programming marks a significant evolution in software development practices. This section explores how AI-driven tools transform programming workflows, enhance productivity, and introduce new paradigms for developers. The discussion is structured around several key themes: improved productivity and efficiency, collaborative coding, the evolving role of developers, and the challenges and opportunities posed by AI integration.

6.1.1 Enhanced Productivity and Efficiency

AI tools such as GitHub Copilot, TabNine, and ChatGPT have demonstrated significant potential in enhancing productivity and efficiency in software development. These tools can automate repetitive coding tasks, generate boilerplate code, and provide real-time suggestions, reducing developers' time on routine activities.

For instance, GitHub Copilot's ability to quickly generate code snippets and entire functions based on natural language prompts allows developers to focus more on complex problem-solving and less on manual coding. This acceleration of the development process can lead to faster project completion times and increased throughput for development teams.

Key Benefits:

- **Time Savings:** Automating repetitive tasks frees up time for developers to tackle more complex and creative aspects of development.
- **Consistency:** AI tools help maintain consistency in code quality and adherence to coding standards across large codebases.
- **Error Reduction:** Real-time code suggestions and corrections reduce the likelihood of syntax and logic errors, enhancing overall code reliability.

6.1.2 Collaborative Coding

Integrating AI tools raises a new level of collaboration within development teams. AI-driven pair programming, where a developer works alongside an AI assistant, enhances the collaborative coding experience.

GitHub Copilot acts as a virtual pair programmer, offering suggestions and improvements as the developer writes code. This interaction speeds up the coding process and serves as a learning tool for developers, exposing them to new coding techniques and best practices. Like GitHub Copilot, TabNine provides context-aware code completions that help developers understand and adopt consistent coding patterns within their teams.

These tools bring benefits like knowledge sharing, where the AI tools introduce developers to new libraries, frameworks, and coding practices, fostering continuous learning and improvement and enhanced review processes where AI-generated code can be a starting point for code reviews, allowing human developers to focus on refining and optimizing rather than writing from scratch.

6.1.3 Evolving Role of Developers

As AI tools become more integrated into the development process, the role of developers is evolving. Rather than replacing developers, AI augments their capabilities, allowing them to focus on higher-order tasks that require human creativity and critical thinking.

Developers can shift their focus from writing code to designing and architecting solutions, leveraging AI to handle the implementation details. With AI handling much of the routine coding, developers can spend more time on code quality assurance, testing, and validation, ensuring robust and reliable software. Finally, AI tools provide opportunities for developers to learn and adapt to new technologies and methodologies through interaction and feedback from the AI.

6.1.4 Challenges and Opportunities

While integrating AI into programming offers numerous benefits, it also presents challenges that must be addressed to maximize its potential.

Challenges like dependency on AI, when a developer is over-reliant on AI tools, can lead to skill atrophy, becoming less proficient in manual coding and problem-solving. Ensuring the security and privacy of code and data processed by AI tools is crucial, especially in sensitive and regulated industries. Using AI in programming also raises ethical questions regarding unfairness, intellectual property, and the transparency of AI-generated code.

Even with these challenges, using AI tools offers multiple opportunities. There is significant potential for innovation in developing more advanced and specialized AI tools tailored to

specific programming languages and frameworks. Educational institutions and organizations can create training programs that focus on using AI tools effectively, preparing the next generation of developers for AI-augmented workflows. AI-driven development introduces new paradigms, such as AI-assisted debugging, automated documentation generation, and intelligent code refactoring, which can transform traditional development practices.

6.1.5 Summary

The integration of AI in programming is reshaping the software development landscape, offering enhanced productivity, encouraging collaboration, and evolving the role of developers. While challenges exist, the opportunities presented by AI-driven tools are vast, promising a future where human ingenuity and AI capabilities work together to drive innovation in software development.

6.2 Limitations of the Study

While this research provides valuable insights into the integration and effectiveness of AI-driven code generation tools, it is essential to acknowledge the limitations that may impact the generalizability and applicability of the findings. Recognizing these limitations helps contextualize the results and highlights areas for future investigation.

6.2.1 Scope of Tools Evaluated

The study focused on a select set of AI-driven code generation tools: GitHub Copilot, TabNine, ChatGPT (3.5, 4.0, and 4o), and Custom GPTs. While these tools represent some of the most advanced and widely used options available, excluding other notable tools like Amazon CodeWhisperer and Snyk Code because Celfocus has no interest in the specific functionalities of each tool, limits the comprehensiveness of the evaluation. Future studies should include a broader range of tools for a more extensive comparative analysis.

6.2.2 Sample Size and Diversity

The evaluation relied on feedback and data from a small and specific sample of developers within Celfocus. This limited sample size and the homogeneity of the participants' backgrounds may not fully capture the diversity of experiences and perspectives on the broader developer community. Expanding the study to include a more extensive and diverse sample of developers from different organizations and industries would enhance the validity and reliability of the findings.

6.2.3 Focus on Java Microservices

The study specifically targeted the use of AI tools in generating code for Java microservices within the Spring Boot framework. While this focus allowed for a detailed and in-depth analysis, it may not fully reflect the tools' performance and utility in other programming languages, frameworks, or development environments. Future research should explore the applicability of these AI tools across a broader range of contexts and technologies.

6.2.4 Rapid Evolution of AI Technologies

AI-driven code generation tools are rapidly evolving, with frequent updates and new features being introduced regularly. The findings of this study are based on the versions of the tools available at the time of the research. As these tools continue to develop, their capabilities, performance, and integration options may change, potentially impacting the relevance of the current findings. Continuous monitoring and re-evaluation of these tools are necessary to keep the insights current.

6.2.5 Context-Specific Findings

The findings of this study are context-specific and may not be completely applicable. The integration and effectiveness of AI tools can vary significantly depending on organizational culture, existing development practices, and project-specific needs. Generalizing the results beyond the context of Celfocus and similar organizations should be done cautiously, and further studies in diverse settings are necessary to validate the conclusions.

7 Conclusion

This chapter concludes the research by delving into the comprehensive explanation of how each objective was fulfilled, synthesizing the findings and discussing their practical implications for businesses and engineers. It encapsulates the transformative impact of AI-driven code generation tools on software development practices, highlighting the opportunities and challenges associated with their integration. The discussion aims to provide actionable insights that can guide organizations and developers in leveraging AI to enhance productivity, improve code quality, and foster innovation.

7.1 Objectives Fulfillment

The study comprehensively reviewed the literature on AI-driven code generation, mainly focusing on Java microservices using the Spring Boot framework. This review identified several effective techniques, including GitHub Copilot, ChatGPT, and TabNine, while highlighting significant knowledge gaps in integrating AI-generated code into established development workflows and the necessity for human oversight to refine AI outputs.

The detailed comparative analysis of various AI tools for code generation revealed that GitHub Copilot, Custom GPTs, and ChatGPT 4.0/4o performed exceptionally well in speed and accuracy. The evaluation underscored GitHub Copilot's efficiency in integrating database connections and configuring Jenkins pipelines, while Custom GPTs provided the most accurate code generation with minimal errors.

The performance evaluation demonstrated that AI tools like GitHub Copilot and Custom GPTs significantly enhanced code generation speed and accuracy compared to traditional methods. GitHub Copilot averaged 7 minutes for test implementation, and Custom GPTs consistently produced accurate and efficient code with fewer errors across various tasks.

When implementing a functional case study, it was decided to divide the case study into multiple smaller parts. Instead of measuring the efficiency of making a microservice, it was decided that it was better to measure for developing APIs, making tests, integrating database connections, altering swagger documentation, configuring Jenkins pipelines, and improving existing code to validate the practical application of these AI tools. The case studies confirmed the tools' effectiveness in generating code that meets functional requirements, though human intervention was necessary to align the AI outputs with specific project standards and goals.

Comparative analysis with traditional code generation methods highlighted that AI-driven approaches significantly reduced development time and improved code quality. However, the study also emphasized the importance of human oversight to ensure the AI-generated code adheres to best practices and project specifications.

The AI tools demonstrated varying levels of autonomy across different development stages. While they excelled in generating user stories and initial code, human intervention was crucial for refining and integrating observability components, API documentation, and CI/CD processes. The tools showed potential in automating routine tasks but required human expertise for complex decision-making.

The analysis of results indicated that AI tools substantially improve productivity and code quality, particularly for routine and repetitive tasks. AI-generated code requires fewer manual adjustments and errors than human-generated code, validating the practical benefits of integrating AI into the development workflow.

The study proposed several improvements for AI-based code generation, including enhancing AI tools' usability and integration capabilities. Future research should focus on expanding the range of evaluated tools, conducting long-term impact studies, and exploring advanced security and ethical considerations in AI-driven development.

Finally, the research identified several security and privacy concerns associated with AI-driven code generation. It emphasized the need for robust security assessments and the development of best practices to safeguard sensitive information and ensure the integrity of AI-generated code. Addressing these issues is critical for the widespread adoption of AI in software development.

7.2 Summary of Findings

The integration of AI in software development, with a specific focus on Java microservices, signifies a transformative era in the field. This research comprehensively explores AI's multifaceted impact, ranging from its influence on programming education to its role in redefining traditional coding practices. AI methodologies introduce efficiency and acceleration in development processes and bring unique challenges, including ensuring code functionality, adherence to established coding standards, and the crucial need for human oversight in refining AI-generated outputs.

The study also underscores the financial and resource challenges, particularly for smaller firms. These firms grapple with the complexities of adopting AI, hindered by limited access to specialized knowledge and resources. This disparity highlights a broader issue in the technology sector, where the pace of technological advancements exacerbates the divide between giant corporations and smaller firms.

Research indicates a significant impact of AI tools on programming education. Studies show a range of responses from university instructors adapting to AI tools like ChatGPT and GitHub Copilot, from banning these tools to embracing them as essential educational aids. Additionally, student perspectives reveal both productivity benefits and challenges, including concerns about over-reliance on AI for coding.

Empirical evaluations of AI tools such as GitHub Copilot, Amazon CodeWhisperer, and ChatGPT reveal high success rates in generating valid code but also highlight limitations in producing entirely correct and robust code. Method names significantly impact the robustness of pre-trained code generation models, and various methodologies, such as RADAR, have been proposed to address these vulnerabilities.

The protection of intellectual property (IP) in AI-generated code is crucial. Techniques like ToSyn for watermarking significant language model-based code generation outputs demonstrate effective methods to safeguard IP by embedding unique identifiers in generated code. This protection is essential for maintaining innovation and securing investments in AI technologies.

Security remains a critical concern in AI-generated code. The development of datasets like SecurityEval to evaluate the security robustness of machine learning-based code generation tools underscores the need for thorough security evaluations. Studies on ChatGPT's ability to identify and fix code vulnerabilities highlight AI's potential and limitations in enhancing software security education.

Innovative approaches such as constraint-based code generation and symbolic machine learning for automating code generation in Model-Driven Engineering (MDE) represent significant advancements. These methods promise to streamline code generation processes and enhance productivity. Empirical studies on AI pair programmers like GitHub Copilot demonstrate substantial productivity gains, particularly for less experienced developers.

The necessity for ongoing studies in this domain is emphasized. Critical areas for future exploration include enhancing AI-generated code's robustness, security, and ethical considerations. As AI continues to evolve, it is paramount to maintain a delicate balance between technological innovation and moral responsibility. This equilibrium is vital for harnessing AI's full potential to augment software development processes and educational frameworks.

The implications of this study extend beyond immediate technical advancements, impacting how developers, educators, and businesses approach software creation and learning. The vision is to pave the way for an era where AI accelerates and enriches software development practices, fostering an environment of innovation, inclusivity, and continuous learning.

In conclusion, this thesis presents a nuanced perspective on AI's role as a development accelerator in Java microservices. It lays a foundation for future research and development, aiming for a future where AI effectively bridges the gap between rapid technological growth and sustainable, ethical software development practices. The findings highlight the transformative potential of AI in software development, emphasizing the need for continued research, ethical considerations, and the development of robust, secure, and reliable AI-driven tools.

7.3 Practical Implications

Integrating AI in software development has significant practical implications for businesses and engineers. By automating routine tasks, improving accuracy, and accelerating development processes, AI-driven tools can enhance the efficiency and effectiveness of software development workflows. However, realizing these benefits requires a strategic approach to adopting and integrating AI technologies.

7.3.1 For Businesses

Businesses can benefit from incorporating AI-driven code-generation tools into their development processes. However, they must navigate several strategic and operational considerations to maximize these benefits.

AI tools like GitHub Copilot and ChatGPT can automate repetitive coding tasks, freeing developers to focus on more complex and innovative aspects of projects. This can lead to significant productivity gains and faster project completion times.

AI tools can help lower development costs by reducing the amount of manual coding required and minimizing errors early in the development process. Businesses can also save on training costs by using AI to upskill their developers in real-time.

Early adopters of AI-driven code generation tools can gain a competitive edge by bringing products to market faster and maintaining higher code quality. This advantage is particularly relevant in highly competitive industries where speed and innovation are critical.

Businesses must invest in the necessary infrastructure and training to support the effective use of AI tools. This includes integrating these tools into existing development workflows and ensuring developers are proficient in using them. These businesses must also address AI-generated code's ethical implications and security risks, implement robust security measures, and maintain transparency in AI operations.

Organizations should also evaluate the scalability of AI tools to ensure they can handle large-scale projects and adapt to growing business needs.

7.3.2 For Engineers

Integrating AI-driven code generation tools significantly shifts engineers' development paradigms. These tools can augment their capabilities, allowing them to work more efficiently and focus on higher-order tasks.

AI tools can provide real-time code suggestions, automate testing, and improve documentation, making the development process smoother and more enjoyable. This enhanced experience can lead to higher job satisfaction and reduced burnout.

Working alongside AI tools can expose engineers to new coding techniques and best practices, facilitating continuous learning and professional growth. Engineers can leverage these tools to stay updated with the latest advancements in programming.

AI tools can help engineers produce cleaner, more reliable code by providing real-time feedback and identifying potential errors early in development. This leads to improved software quality and reduced maintenance efforts.

Engineers need to adapt to AI's evolving role in development. This includes being open to learning new tools and techniques and integrating AI-driven workflows into their daily practices. Effective collaboration with AI tools requires a synergistic approach where engineers leverage AI for routine tasks while focusing on creative problem-solving and innovation. Also, engineers must be mindful of the ethical implications of using AI in development. This includes ensuring fairness, avoiding bias, and maintaining the transparency and integrity of AI-generated code.

7.3.3 Summary

The practical implications of integrating AI in software development are profound, offering numerous benefits for businesses and engineers. By strategically adopting AI tools, companies can achieve significant productivity gains, cost savings, and competitive advantages. Engineers can enhance their development experience, continuously learn, and improve code quality by effectively collaborating with AI. Addressing the associated challenges and considerations is crucial to realizing these benefits and harnessing AI's full potential in transforming software development practices.

8 Future Work

This chapter outlines the prospective directions for future research and highlights emerging trends in AI-driven software development. It aims to provide a roadmap for continued exploration and innovation in integrating AI into programming workflows.

8.1 Recommendations for Future Research

Several vital areas warrant further investigation to advance the integration and utility of AI-driven code-generation tools in software development. These recommendations are designed to address current gaps, explore new dimensions, and enhance AI's overall understanding and effectiveness in programming.

8.1.1 Expanding the Range of Evaluated Tools

Future research should include a broader array of AI-driven code-generation tools by exploring emerging and less-studied AI tools such as Amazon CodeWhisperer, Snyk Code, and other innovative solutions. This will provide a more comprehensive understanding of the landscape and help identify niche capabilities and limitations. Future research should also assess the performance and applicability of these tools across various programming languages (e.g., Python, JavaScript, C#) and frameworks (e.g., Django, React, .NET). This approach will determine their versatility and broader utility.

8.1.2 Redo the Survey

Following the distribution of AI-powered code assistance tool licenses, it is recommended that the company conduct a follow-up survey to assess the impact and gather insights on the developers' evolving perceptions and experiences. This subsequent survey should aim to capture changes in productivity, code quality, security concerns, and overall user satisfaction after integrating these tools into their daily workflows.

By comparing the results of this follow-up survey with the initial survey, the company can gain valuable insights into the effectiveness of the AI tools and make informed decisions about future investments and improvements. This iterative feedback loop will help ensure that the tools continue to meet the developers' needs and contribute positively to the company's development processes.

8.1.3 Enhancing Sample Diversity and Size

Engage a broader and more diverse group of developers from different industries, experience levels, and geographical regions to obtain a more representative understanding of AI tool performance. This will help capture broader insights and ensure findings are more generalizable. It should also conduct studies over extended periods to observe the long-term effects of AI tool usage on developer productivity, skill development, and project outcomes.

8.1.4 In-Depth Analysis of Long-Term Impacts

Investigate the sustained impacts of AI integration in development environments by exploring how continuous use of AI tools influences developers' skills, especially concerning manual coding proficiency and problem-solving abilities, and examining the effects of AI-generated code on long-term project maintenance, including codebase scalability, refactoring requirements, and technical debt management.

8.1.5 Advanced Security and Ethical Research

Focus on AI-driven code generation's security and ethical dimensions. Perform in-depth security assessments of AI-generated code to identify vulnerabilities. Develop best practices for secure AI implementation. Establish ethical guidelines for AI usage in software development, ensuring fairness, transparency, and accountability by addressing bias in AI-generated code and safeguarding intellectual property rights.

8.1.6 Optimizing Workflow Integration

Improve the integration of AI tools into existing development workflows by developing methodologies for integrating AI tools more effectively into continuous integration and continuous deployment (CI/CD) pipelines, enhancing overall workflow efficiency and focusing on improving the usability and user experience of AI tools, making them more accessible and intuitive for developers of varying expertise levels.

8.1.7 Educational Initiatives and Impact

Design educational programs that integrate AI tools, preparing students for future industry demands and fostering a new generation of developers skilled in AI-augmented development, and investigate how AI tools influence learning outcomes, particularly in understanding fundamental programming concepts and enhancing problem-solving skills to explore the role of AI in programming education.

8.1.8 Exploring AI-Assisted Collaborative Development

Research the dynamics of AI-assisted pair programming, where AI tools collaborate with human developers in real-time, assess the impact on team productivity and learning, and examine how AI tools can support distributed development teams, facilitating better communication, code consistency, and project management across different locations.

8.1.9 Summary

These recommendations for future research aim to address the current study's limitations and expand the understanding of AI's role in software development. By exploring these areas, researchers can contribute to creating more effective, secure, and ethically sound AI-driven development tools, enhancing the productivity and capabilities of software developers worldwide.

8.2 Future Trends to Watch

As AI continues to evolve and integrate into software development, several emerging trends are poised to shape the future of programming. Keeping an eye on these trends will help developers, businesses, and researchers stay ahead of the curve and leverage AI's full potential.

8.2.1 Advanced Natural Language Processing

Natural Language Processing (NLP) is at the core of many AI-driven code-generation tools. Future advancements in NLP will lead to AI tools with better understanding and generating code based on more nuanced and complex natural language inputs, making them even more intuitive for developers. Tools will also be able to maintain context more effectively over extended interactions, improving the relevance and accuracy of code suggestions.

8.2.2 AI-Driven Code Refactoring and Optimization

AI tools are expected to become more proficient in generating code and optimizing and refactoring it. AI will assist in restructuring existing code to improve readability, performance, and maintainability without altering its functionality. AI tools automatically suggest and implement optimizations, making code more efficient and reducing resource consumption.

8.2.3 Integration of AI in DevOps

The role of AI in DevOps is set to expand, leading to more streamlined and automated workflows. AI will enhance CI/CD pipelines by automating more complex tasks such as

environment setup, testing, and deployment, improving overall efficiency and reliability. AI-driven analytics will provide insights into potential issues before they arise, allowing teams to address problems and optimize workflows proactively.

8.2.4 AI-Augmented Debugging and Testing

Debugging and testing are critical development lifecycle phases where AI can provide significant benefits. AI tools will become more adept at identifying and fixing bugs automatically, reducing the time developers spend on debugging. AI will also enhance testing processes by automatically generating test cases, identifying edge cases, and predicting potential points of failure, ensuring more robust and reliable software.

8.2.5 Summary

The future of AI in software development is filled with exciting possibilities. These trends will shape how software is developed, tested, and maintained, from advanced NLP to AI-augmented debugging and testing code optimization. By staying attuned to these trends, developers, businesses, and researchers can harness AI's transformative potential, driving innovation and efficiency in the ever-evolving software development landscape.

References

- Bhati, N., 2023. AI in Software Development: Trends, Benefits and Cost. Artificial Intelligence Development Company | Appquipo. URL <https://appquipo.com/blog/ai-in-software-development/> (accessed 12.12.23).
- Cerutti, N., 2023. Copilot VSCode Extension causes high cpu usage · community · Discussion #53846 · GitHub [WWW Document]. URL <https://github.com/orgs/community/discussions/53846> (accessed 6.30.24).
- Clarisó, R., Cabot, J., 2023. Model-Driven Prompt Engineering, in: 2023 ACM/IEEE 26th International Conference on Model Driven Engineering Languages and Systems (MODELS). Presented at the 2023 ACM/IEEE 26th International Conference on Model Driven Engineering Languages and Systems (MODELS), IEEE, Västerås, Sweden, pp. 47–54. <https://doi.org/10.1109/MODELS58315.2023.00020>
- Code faster and better with GitHub Copilot's new features in Visual Studio, 2024. . The GitHub Blog. URL <https://github.blog/changelog/2024-01-30-code-faster-and-better-with-github-copilots-new-features-in-visual-studio/> (accessed 5.18.24).
- Configuring content exclusions for GitHub Copilot [WWW Document], 2024. . GitHub Docs. URL <https://docs.github.com/en/copilot/managing-github-copilot-in-your-organization/configuring-content-exclusions-for-github-copilot> (accessed 6.5.24).
- Curry, D., 2024. ChatGPT Revenue and Usage Statistics (2024) [WWW Document]. Business of Apps. URL <https://www.businessofapps.com/data/chatgpt-statistics/> (accessed 5.19.24).
- Davenport, T.H., Mittal, N., 2022. How companies can prepare for the coming “AI-first” world. *Strategy & Leadership* 51, 26–30. <https://doi.org/10.1108/SL-11-2022-0107>
- Degges, R., 2024. Copilot amplifies insecure codebases by replicating vulnerabilities in your projects [WWW Document]. Snyk. URL <https://snyk.io/blog/copilot-amplifies-insecure-codebases-by-replicating-vulnerabilities/> (accessed 5.19.24).
- Dong, Y., Jiang, X., Jin, Z., Li, G., 2024. Self-collaboration Code Generation via ChatGPT. Features - CodeWhisperer [WWW Document], n.d. URL <https://docs.aws.amazon.com/codewhisperer/latest/userguide/features.html> (accessed 5.18.24).
- GitHub Copilot overview [WWW Document], 2024. URL <https://code.visualstudio.com/docs/copilot/overview> (accessed 5.18.24).
- IEEE Code of Ethics [WWW Document], 2020. URL <https://www.ieee.org/about/corporate/governance/p7-8.html> (accessed 12.15.23).
- Lau, S., Guo, P., 2023a. From “Ban It Till We Understand It” to “Resistance is Futile”: How University Programming Instructors Plan to Adapt as More Students Use AI Code Generation and Explanation Tools such as ChatGPT and GitHub Copilot, in: *Proceedings of the 2023 ACM Conference on International Computing Education Research V.1*. Presented at the ICER 2023: ACM Conference on International Computing Education Research, ACM, Chicago IL USA, pp. 106–121. <https://doi.org/10.1145/3568813.3600138>
- Lau, S., Guo, P., 2023b. From “Ban It Till We Understand It” to “Resistance is Futile”: How University Programming Instructors Plan to Adapt as More Students Use AI Code Generation and Explanation Tools such as ChatGPT and GitHub Copilot, in: *Proceedings of the 2023 ACM Conference on International Computing Education Research V.1*. Presented at the ICER 2023: ACM Conference on International

- Computing Education Research, ACM, Chicago IL USA, pp. 106–121.
<https://doi.org/10.1145/3568813.3600138>
- Li, J., Meland, P., Notland, J., Storhaug, A., Tysse, J., 2024. Evaluating the Impact of ChatGPT on Exercises of a Software Security Course [WWW Document]. URL <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=10304857> (accessed 5.19.24).
- Li, Z., Wang, C., Wang, S., Gao, C., 2023. Protecting Intellectual Property of Large Language Model-Based Code Generation APIs via Watermarks, in: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS '23. Association for Computing Machinery, New York, NY, USA, pp. 2336–2350.
<https://doi.org/10.1145/3576915.3623120>
- Liang, J.T., Yang, C., Myers, B.A., 2023. A Large-Scale Survey on the Usability of AI Programming Assistants: Successes and Challenges.
- Lim, R., Wu, M., Miller, L., 2021. Customizing GPT-3 for your application [WWW Document]. URL <https://openai.com/index/customizing-gpt-3/> (accessed 5.18.24).
- Lin, Y., Liu, S., Yang, H., Wu, H., Jiang, B., 2021. Improving stock trading decisions based on pattern recognition using machine learning technology. PLOS ONE 16, e0255558.
<https://doi.org/10.1371/journal.pone.0255558>
- Losbichler, H., Lehner, O.M., 2021. Limits of artificial intelligence in controlling and the ways forward: a call for future accounting research. Journal of Applied Accounting Research 22, 365–382. <https://doi.org/10.1108/JAAR-10-2020-0207>
- Lozano, R.C., Blindell, G.H., Carlsson, M., Drejhammar, F., Schulte, C., 2013. Constraint-based code generation, in: Proceedings of the 16th International Workshop on Software and Compilers for Embedded Systems, M-SCOPES '13. Association for Computing Machinery, New York, NY, USA, pp. 93–95. <https://doi.org/10.1145/2463596.2486155>
- Maheshwari, P., 2023. ChatGPT 4.0 vs 3.5: The Battle of AI Titans. Open AI Master. URL <https://openaimaster.com/chatgpt-4-0-vs-3-5/> (accessed 5.18.24).
- Mamgain, D., 2024. ChatGPT 3.5 vs 4 vs 4o - Key Differences to Consider. Kommunicate Blog. URL <https://www.kommunicate.io/blog/chatgpt-4-vs-chatgpt-3-5-key-differences/> (accessed 6.30.24).
- McDermott, M.B.A., Nestor, B., Argaw, P., Kohane, I., 2023. Event Stream GPT: A Data Pre-processing and Modeling Library for Generative, Pre-trained Transformers over Continuous-time Sequences of Complex Events.
- Mudgal, P., Wouhaybi, R., 2023. An Assessment of ChatGPT on Log Data.
- OpenAI, 2024. Hello GPT-4o [WWW Document]. URL <https://openai.com/index/hello-gpt-4o/> (accessed 5.18.24).
- OpenAI, n.d. Introducing ChatGPT Enterprise [WWW Document]. URL <https://openai.com/index/introducing-chatgpt-enterprise/> (accessed 6.5.24).
- Peng, S., Kalliamvakou, E., Demirer, M., 2024. The Impact of AI on Developer Productivity: Evidence from GitHub Copilot [WWW Document]. ar5iv. URL <https://ar5iv.labs.arxiv.org/html/2302.06590> (accessed 5.18.24).
- Plans & Pricing | Tabnine: The AI code assistant that you control [WWW Document], n.d. . Tabnine. URL <https://www.tabnine.com/pricing/> (accessed 6.5.24).
- Qammar, A., Wang, H., Ding, J., Naouri, A., Daneshmand, M., Ning, H., 2023. Chatbots to ChatGPT in a Cybersecurity Space: Evolution, Vulnerabilities, Attacks, Challenges, and Future Recommendations.
- Regulation 2016/679 - RGPD [WWW Document], 2016. URL https://www.eumonitor.eu/9353000/1/j4nkv6yhcbpeywk_j9vvik7m1c3gyxp/vk3t7p3lbczq (accessed 12.15.23).

- Reilly, J., 2023. A cost breakdown of artificial intelligence in 2023 [WWW Document]. Akkio. URL <https://www.akkio.com/post/a-cost-breakdown-of-artificial-intelligence-in-2023> (accessed 12.12.23).
- Saka, A., Taiwo, R., Saka, N., Salami, B.A., Ajayi, S., Akande, K., Kazemi, H., 2024. GPT models in construction industry: Opportunities, limitations, and a use case validation. *Developments in the Built Environment* 17, 100300. <https://doi.org/10.1016/j.dibe.2023.100300>
- Siddiq, M.L., Santos, J.C.S., 2022. SecurityEval dataset: mining vulnerability examples to evaluate machine learning-based code generation techniques, in: *Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security, MSR4P&S 2022*. Association for Computing Machinery, New York, NY, USA, pp. 29–33. <https://doi.org/10.1145/3549035.3561184>
- Snyk Code | Code Security Analysis and Fixes - Developer First SAST [WWW Document], n.d. . Snyk. URL <https://snyk.io/product/snyk-code/> (accessed 5.18.24).
- Sridhara, G., G., R.H., Mazumdar, S., 2023. ChatGPT: A Study on its Utility for Ubiquitous Software Engineering Tasks.
- Sun, Z., Du, X., Song, F., Wang, S., Ni, M., Li, L., 2023. Don't Complete It! Preventing Unhelpful Code Completion for Productive and Sustainable Neural Code Completion Systems.
- Tabnine | Total code privacy & zero data retention [WWW Document], n.d. . Tabnine. URL <https://www.tabnine.com/code-privacy/> (accessed 5.19.24).
- Takerngsaksiri, W., Warusavitarn, C., Yaacoub, C., Hou, M., Tantithamthavorn, C., 2023. Students' Perspective on AI Code Completion: Benefits and Challenges [WWW Document]. ar5iv. URL <https://ar5iv.labs.arxiv.org/html/2311.00177> (accessed 5.19.24).
- Vaithilingam, P., Glassman, E.L., Groenwegen, P., Gulwani, S., Henley, A.Z., Malpani, R., Pugh, D., Radhakrishna, A., Soares, G., Wang, J., Yim, A., 2023. Towards More Effective AI-Assisted Programming: A Systematic Design Exploration to Improve Visual Studio IntelliCode's User Experience, in: *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. Presented at the 2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), IEEE, Melbourne, Australia, pp. 185–195. <https://doi.org/10.1109/ICSE-SEIP58684.2023.00022>
- Valerie, J., 2023. Tabnine [WWW Document]. Selected AI. URL <https://www.selectedai.com/tabnine/> (accessed 5.18.24).
- Wilson-Thomas, M., 2023. Simplified Code Refinement and Debugging with GitHub Copilot Chat [WWW Document]. Visual Studio Blog. URL <https://devblogs.microsoft.com/visualstudio/simplified-code-refinement-and-debugging-with-github-copilot-chat/> (accessed 5.18.24).
- Xue, Q., 2023. Automating Code Generation for MDE Using Machine Learning, in: *Proceedings of the 45th International Conference on Software Engineering: Companion Proceedings, ICSE '23*. IEEE Press, Melbourne, Victoria, Australia, pp. 221–223. <https://doi.org/10.1109/ICSE-Companion58688.2023.00060>
- Yahav, E., 2021. Tabnine memory and CPU usage [WWW Document]. Tabnine. URL <https://www.tabnine.com/blog/tabnine-memory-and-cpu-usage/> (accessed 6.30.24).
- Yang, G., Zhou, Y., Yang, W., Yue, T., Chen, X., Chen, T., 2023. How Important are Good Method Names in Neural Code Generation? A Model Robustness Perspective. *ACM Trans. Softw. Eng. Methodol.* <https://doi.org/10.1145/3630010>
- Yetiştirilen, B., Özsoy, I., Ayerdem, M., Tüzün, E., 2023a. Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT. <https://doi.org/10.48550/arXiv.2304.10778>

- Yetiştiren, B., Özsoy, I., Ayerdem, M., Tüzün, E., 2023b. Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT.
- Yetiştiren, B., Özsoy, I., Ayerdem, M., Tüzün, E., 2023c. Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT.
- Yetistiren, B., Ozsoy, I., Tuzun, E., 2022. Assessing the quality of GitHub copilot's code generation, in: Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering, PROMISE 2022. Association for Computing Machinery, New York, NY, USA, pp. 62–71.
<https://doi.org/10.1145/3558489.3559072>
- Zhang, B., Liang, P., Zhou, X., Ahmad, A., Waseem, M., 2023. Demystifying Practices, Challenges and Expected Features of Using GitHub Copilot.

Annexes

Annex 1 - Design and Implementation

	January			February				March				April		
Task	15-19	22-26	29-02	05-09	12-16	19-23	26-01	04-08	11-15	18-22	25-29	01-05	08-12	15-19
Develop a detailed project plan, outlining the design and implementation stages.														
Define specifications for the microservice.														
Set up the development environment and tools necessary for the Spring Boot framework.														
Begin coding basic structures and functionalities of the microservice.														
Implement database connectivity and external REST service consumption.														
Develop the AI-based code generation model.														
Integrate generated code into the microservice architecture.														
Begin unit testing and documentation of the implementation process.														
Finalize core development; ensure all functionalities are implemented.														
Refine code through iterative testing and feedback.														
Implement observability components and API documentation.														
Conduct comprehensive code reviews and static code analysis.														

Annex 2 - Experimentation and Evaluation

	April			May				June			
Task	15-19	22-26	29-03	06-10	13-17	20-24	27-31	03-07	10-14	17-21	24-28
Define evaluation criteria and metrics for the AI-generated code.											
Develop and document detailed test cases for the functional development case.											
Run tests; collect data on performance, accuracy, and efficiency.											
Analyze initial test data; refine the microservice based on results.											
Compare AI-generated code to traditional methods; document findings.											
Draft the analysis and results sections for the thesis.											
Write the discussion, conclusion, and recommendations for future research.											
Compile the full thesis draft; submit to the advisor for preliminary review.											

Annex 3 - Effective Prompt for Test Implementation



Write a JUnit test for a method that calculates the factorial of a number.



To write a JUnit test for a method that calculates the factorial of a number, you'll need to follow these steps:

1. **Create the method for calculating factorial:** Ensure you have a method in your code that calculates the factorial of a given number.
2. **Write the JUnit test:** Create a test class and write test methods to verify the correctness of your factorial method.

Here is an example:

Factorial Method

```
java Copy code  
  
public class FactorialCalculator {  
  
    public static long factorial(int number) {  
        if (number < 0) {  
            throw new IllegalArgumentException("Number must be non-negative.");  
        }  
        long result = 1;  
        for (int i = 1; i <= number; i++) {  
            result *= i;  
        }  
        return result;  
    }  
}
```

JUnit Test

```
java Copy code  
  
import org.junit.jupiter.api.Test;  
import static org.junit.jupiter.api.Assertions.*;  
  
public class FactorialCalculatorTest {  
  
    @Test  
    public void testFactorialZero() {  
        assertEquals(1, FactorialCalculator.factorial(0));  
    }  
  
    @Test
```



Annex 4 - Effective Prompt for API Development

Create a RESTful API endpoint in Spring Boot for retrieving user information by user ID.



To create a RESTful API endpoint in Spring Boot for retrieving user information by user ID, follow these steps:

1. Set Up the Spring Boot Project:

- If you haven't already, create a new Spring Boot project using Spring Initializr (<https://start.spring.io/>).
- Add the necessary dependencies: `Spring Web`, `Spring Data JPA`, and `H2 Database` (or any other database of your choice).

2. Create the User Entity:

- Define an entity class to map to your database table.

3. Create the User Repository:

- Define a repository interface to handle data access.

4. Create the User Service:

- Implement the service layer to encapsulate the business logic.

5. Create the User Controller:

- Implement the controller to handle HTTP requests.

6. Configure the Application Properties:

- Set up the necessary configurations for your database in `application.properties`.

Here's the implementation:

Step 1: Set Up the Spring Boot Project

In your `pom.xml` (for Maven), add the following dependencies:

```
xml Copy code  
  
<dependencies>  
  <dependency>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starter-web</artifactId>  
  </dependency>  
  <dependency>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starter-data-jpa</artifactId>  
  </dependency>  
  <dependency>  
    <groupId>com.h2database</groupId>  
    <artifactId>h2</artifactId>  
    <scope>runtime</scope>  
  </dependency>  
</dependencies>
```

Step 2: Create the User Entity

Create a `User` class annotated with `@Entity`.

```
java Copy code

package com.example.demo.entity;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String email;

    // Getters and setters
    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }
}
```

Step 3: Create the User Repository

Create an interface `UserRepository` that extends `JpaRepository`.

```
java Copy code

package com.example.demo.repository;

import com.example.demo.entity.User;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface UserRepository extends JpaRepository<User, Long> {
}
```

Step 4: Create the User Service

Create a 'UserService' class to handle the business logic.

```
java Copy code  
  
package com.example.demo.service;  
  
import com.example.demo.entity.User;  
import com.example.demo.repository.UserRepository;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Service;  
  
import java.util.Optional;  
  
@Service  
public class UserService {  
  
    @Autowired  
    private UserRepository userRepository;  
  
    public Optional<User> getUserById(Long id) {  
        return userRepository.findById(id);  
    }  
}
```

Step 5: Create the User Controller

Create a 'UserController' class to handle HTTP requests.

```
java Copy code  
  
package com.example.demo.controller;  
  
import com.example.demo.entity.User;  
import com.example.demo.service.UserService;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.http.ResponseEntity;  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.PathVariable;  
import org.springframework.web.bind.annotation.RequestMapping;  
import org.springframework.web.bind.annotation.RestController;  
  
import java.util.Optional;  
  
@RestController  
@RequestMapping("/api/users")  
public class UserController {  
  
    @Autowired  
    private UserService userService;  
  
    @GetMapping("/{id}")  
    public ResponseEntity<User> getUserById(@PathVariable Long id) {  
        Optional<User> user = userService.getUserById(id);  
        return user.map(ResponseEntity::ok).orElseGet(() -> ResponseEntity.notFound());  
    }  
}
```

Step 6: Configure the Application Properties

In your `src/main/resources/application.properties`, configure the database settings.

```
properties Copy code  
  
spring.datasource.url=jdbc:h2:mem:testdb  
spring.datasource.driverClassName=org.h2.Driver  
spring.datasource.username=sa  
spring.datasource.password=password  
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect  
spring.h2.console.enabled=true  
spring.jpa.show-sql=true  
spring.jpa.hibernate.ddl-auto=update
```

Running the Application

- Run your Spring Boot application.
- Access the endpoint via `http://localhost:8080/api/users/{id}` where `{id}` is the user ID you want to retrieve.

This setup provides a basic RESTful API endpoint to retrieve user information by user ID using Spring Boot.

Annex 5 - Ineffective Prompt with Vague Instructions

Create a method to handle data.

Creating a method to handle data depends on the specific requirements of the task. I'll outline a general-purpose data handling method in Python that can be adapted for various use cases. This method will include loading data, processing it, and saving the processed data.

Let's create a method that:

1. Loads data from a file (e.g., CSV).
2. Processes the data (e.g., cleans it, performs some analysis).
3. Saves the processed data to a new file.

Here's a basic example:

```
python Copy code

import pandas as pd

def handle_data(input_file, output_file, process_func):
    """
    Handle data by loading it from a file, processing it, and saving it to a new file.

    Parameters:
    - input_file (str): The path to the input file (e.g., a CSV file).
    - output_file (str): The path to the output file where the processed data will be saved.
    - process_func (function): A function that defines the data processing steps.

    Returns:
    - None
    """
    # Load data
    data = pd.read_csv(input_file)
    print(f"Data loaded from {input_file}")

    # Process data
    processed_data = process_func(data)
    print("Data processing complete")

    # Save processed data
    processed_data.to_csv(output_file, index=False)
    print(f"Processed data saved to {output_file}")

# Example process function
def process_func(data):
    """
    Example process function to clean and analyze data.

    Parameters:
    - data (DataFrame): The input data to be processed.

    Returns:
    - DataFrame: The processed data.
    """
    # Example processing steps
    # 1. Remove rows with missing values
    data_cleaned = data.dropna()

    # 2. Add a new column with some calculated values
    data_cleaned['new_column'] = data_cleaned['existing_column'] * 2

    return data_cleaned

# Example usage
input_file = 'path/to/your/input_file.csv'
output_file = 'path/to/your/output_file.csv'

handle_data(input_file, output_file, process_func)
```

Explanation

1. **handle_data Function:** This function takes three arguments: the input file path, the output file path, and a processing function. It loads the data from the input file, processes it using the given function, and then saves the processed data to the output file.
2. **process_func Function:** This is an example of a processing function that can be passed to `handle_data`. It demonstrates simple data cleaning (removing rows with missing values) and a transformation (adding a new column based on an existing one). You can customize this function to include any data processing steps you need.
3. **Example Usage:** The `handle_data` function is called with the input file path, output file path, and the processing function.

You can expand and modify the `process_func` to include more complex data handling steps like normalization, outlier removal, data transformation, feature extraction, etc. This modular approach allows you to keep the data handling logic flexible and reusable.

Annex 6 - Detailed Prompt

Write a Java program that reads a list of integers from the user, sorts the list in ascending order using the merge sort algorithm, and then prints the sorted list. The program should include the following functionalities:

1. Prompt the user to enter the number of integers.
2. Prompt the user to enter each integer one by one.
3. Use the merge sort algorithm to sort the list.
4. Print the sorted list.
5. Include comments explaining each step of the code.



Here's a Java program that accomplishes the specified tasks. The program reads a list of integers from the user, sorts it using the merge sort algorithm, and then prints the sorted list. Each step is explained with comments.

```
java Copy code

import java.util.Scanner;

public class MergeSort {

    // Method to merge two subarrays of arr[].
    // First subarray is arr[l..m]
    // Second subarray is arr[m+1..r]
    public static void merge(int arr[], int left, int mid, int right) {
        // Find sizes of two subarrays to be merged
        int n1 = mid - left + 1;
        int n2 = right - mid;

        // Create temporary arrays
        int L[] = new int[n1];
        int R[] = new int[n2];

        // Copy data to temporary arrays
        for (int i = 0; i < n1; ++i)
            L[i] = arr[left + i];
        for (int j = 0; j < n2; ++j)
            R[j] = arr[mid + 1 + j];

        // Initial indexes of first and second subarrays
        int i = 0, j = 0;
```

```

// Initial index of merged subarray array
int k = left;
while (i < n1 && j < n2) {
    if (L[i] <= R[j]) {
        arr[k] = L[i];
        i++;
    } else {
        arr[k] = R[j];
        j++;
    }
    k++;
}

// Copy remaining elements of L[] if any
while (i < n1) {
    arr[k] = L[i];
    i++;
    k++;
}

// Copy remaining elements of R[] if any
while (j < n2) {
    arr[k] = R[j];
    j++;
    k++;
}
}

// Main function that sorts arr[l..r] using merge()
public static void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        // Find the middle point
        int mid = (left + right) / 2;

        // Sort first and second halves
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

        // Merge the sorted halves
        merge(arr, left, mid, right);
    }
}

// Utility function to print an array
public static void printArray(int arr[]) {
    for (int i = 0; i < arr.length; ++i)
        System.out.print(arr[i] + " ");
    System.out.println();
}
}

```

```

public static void main(String args[]) {
    // Create a Scanner object for input
    Scanner scanner = new Scanner(System.in);

    // Prompt the user to enter the number of integers
    System.out.print("Enter the number of integers: ");
    int n = scanner.nextInt();

    // Create an array to store the integers
    int arr[] = new int[n];

    // Prompt the user to enter each integer one by one
    System.out.println("Enter the integers:");
    for (int i = 0; i < n; i++) {
        arr[i] = scanner.nextInt();
    }

    // Close the scanner
    scanner.close();

    // Sort the array using merge sort
    mergeSort(arr, 0, arr.length - 1);

    // Print the sorted array
    System.out.println("Sorted array:");
    printArray(arr);
}
}

```

Explanation of the Code

1. Imports and Class Definition:

- The program imports the `java.util.Scanner` class for reading user input.
- The class `MergeSort` is defined, containing methods for the merge sort algorithm, merging, and printing arrays.

2. Merge Method:

- The `merge` method takes an array and the indices `left`, `mid`, and `right`.
- It merges two subarrays of `arr[]`, the first subarray from `left` to `mid` and the second from `mid+1` to `right`.

3. Merge Sort Method:

- The `mergeSort` method recursively divides the array into two halves and sorts them.
- It finds the middle point, sorts the first and second halves, and then merges them.

4. Print Array Method:

- The `printArray` method prints the elements of the array.

5. Main Method:

- The `main` method handles user interaction.
- It prompts the user to enter the number of integers and then each integer.
- It calls `mergeSort` to sort the array and `printArray` to print the sorted array.

This program ensures that the list of integers is sorted in ascending order using the merge sort algorithm, demonstrating a clear, commented, and modular approach.

Annex 7 - Concise Prompt

Sort a list of integers using merge sort in Java.

Certainly! Below is a Java implementation of the merge sort algorithm to sort a list of integers:

```
java Copy code

public class MergeSort {

    // Main function that sorts the array using merge sort
    public static void mergeSort(int[] array) {
        if (array.length < 2) {
            return;
        }
        int mid = array.length / 2;
        int[] left = new int[mid];
        int[] right = new int[array.length - mid];

        // Dividing the array into two halves
        for (int i = 0; i < mid; i++) {
            left[i] = array[i];
        }
        for (int i = mid; i < array.length; i++) {
            right[i - mid] = array[i];
        }

        // Recursive call to sort the two halves
        mergeSort(left);
        mergeSort(right);

        // Merging the sorted halves
        merge(array, left, right);
    }

    // Function to merge two sorted subarrays
    public static void merge(int[] array, int[] left, int[] right) {
        int i = 0, j = 0, k = 0;
        while (i < left.length && j < right.length) {
            if (left[i] <= right[j]) {
                array[k++] = left[i++];
            } else {
                array[k++] = right[j++];
            }
        }
        while (i < left.length) {
            array[k++] = left[i++];
        }
        while (j < right.length) {
            array[k++] = right[j++];
        }
    }

    // Main method to test the merge sort implementation
    public static void main(String[] args) {
        int[] array = {38, 27, 43, 3, 9, 82, 10};
        System.out.println("Original array:");
        for (int num : array) {
            System.out.print(num + " ");
        }
        System.out.println();

        mergeSort(array);

        System.out.println("Sorted array:");
        for (int num : array) {
            System.out.print(num + " ");
        }
    }
}
```

Explanation

1. **mergeSort Function:**

- This function takes an array and sorts it using the merge sort algorithm.
- If the array has fewer than 2 elements, it is already sorted.
- The array is split into two halves, left and right.
- The `mergeSort` function is called recursively on each half to sort them.

2. **merge Function:**

- This function merges two sorted subarrays into a single sorted array.
- It compares elements from both subarrays and copies the smaller element into the original array.
- After the loop, any remaining elements from the left or right subarray are copied into the original array.

3. **main Method:**

- This is the entry point of the program.
- An array of integers is defined and printed.
- The `mergeSort` function is called to sort the array.
- The sorted array is then printed.

You can compile and run this code to see the merge sort algorithm in action.

Annex 8 - Contextual Prompt

You are building a Java application that requires sorting large datasets efficiently. Implement the merge sort algorithm to sort a list of integers. Make sure the implementation handles edge cases such as an empty list and a list with one element. Include error handling for invalid input and comments explaining the key parts of the algorithm.

🗨️ Sure! Here's a Java implementation of the Merge Sort algorithm, which includes handling for edge cases and error handling for invalid inputs. The code includes comments to explain the key parts of the algorithm.

```
java Copy code

import java.util.Arrays;

public class MergeSort {

    // Function to perform merge sort on an array of integers
    public static void mergeSort(int[] array) {
        // Check for invalid input
        if (array == null) {
            throw new IllegalArgumentException("Input array cannot be null");
        }

        // Check if the array needs to be sorted (non-empty and not a single element)
        if (array.length > 1) {
            // Split the array into two halves
            int middle = array.length / 2;
            int[] left = Arrays.copyOfRange(array, 0, middle);
            int[] right = Arrays.copyOfRange(array, middle, array.length);

            // Recursively sort both halves
            mergeSort(left);
            mergeSort(right);

            // Merge the sorted halves
            merge(array, left, right);
        }
    }

    // Helper function to merge two sorted arrays into a single sorted array
    private static void merge(int[] array, int[] left, int[] right) {
        int i = 0, j = 0, k = 0;

        // Merge elements into the original array in sorted order
        while (i < left.length && j < right.length) {
            if (left[i] <= right[j]) {
                array[k++] = left[i++];
            } else {
                array[k++] = right[j++];
            }
        }

        // Copy any remaining elements from the left array
        while (i < left.length) {
            array[k++] = left[i++];
        }

        // Copy any remaining elements from the right array
        while (j < right.length) {
            array[k++] = right[j++];
        }
    }
}
```

```

// Main method to test the merge sort algorithm
public static void main(String[] args) {
    int[] array = {38, 27, 43, 3, 9, 82, 10};

    System.out.println("Original array: " + Arrays.toString(array));

    // Perform merge sort
    mergeSort(array);

    System.out.println("Sorted array: " + Arrays.toString(array));
}
}

```

Key Parts of the Algorithm:

1. **Input Validation:** The `mergeSort` function first checks if the input array is `null` and throws an `IllegalArgumentException` if it is. This prevents the function from operating on invalid input.
2. **Base Case:** The function checks if the array length is greater than 1. If not, the array is already sorted (as an empty array or a single-element array is considered sorted).
3. **Divide:** The array is split into two halves. The middle index is calculated, and two new arrays (`left` and `right`) are created using `Arrays.copyOfRange`.
4. **Recursion:** The `mergeSort` function is called recursively on both halves (`left` and `right`).
5. **Merge:** The `merge` function is used to combine the two sorted halves back into the original array. The elements from `left` and `right` are compared and merged in sorted order.
6. **Edge Cases:** The algorithm handles edge cases such as empty arrays and single-element arrays naturally due to the base case check. Invalid input (e.g., `null` array) is handled by throwing an exception.

How to Run:

1. Copy the code into a file named `MergeSort.java`.
2. Compile the code using `javac MergeSort.java`.
3. Run the compiled class using `java MergeSort`.

The provided `main` method demonstrates the usage of the `mergeSort` function with an example array. Feel free to modify the `array` variable in the `main` method to test with different datasets.