ARTIFICIAL INTELLIGENCE IN SOFTWARE ENGINEERING: INTEGRATION AND KEY CHALLENGES

Xiaowei Shao [1], Mariko Shibasaki [2] and Ryosuke Shibasaki[1,2]

[1] Department of Engineering, Reitaku University, Kashiwa, Japan
[2] LocationMind, Tokyo, Japan

ABSTRACT

This paper explores the integration of artificial intelligence (AI) into software engineering. It examines how AI can be effectively incorporated throughout the software development lifecycle, encompassing phases like requirement analysis, system design, code development, testing, and software deployment. It highlights the potential benefits of AI-driven software development, such as increased development efficiency, improved software quality, and enhanced performance. The discussion extends to addressing the substantial challenges that accompany the integration of AI within software development frameworks. These include the limitations of current AI technology in achieving complete automation of large software projects, the need to ensure the accuracy and reliability of AI-generated code, complex task decomposition and verification, multi-agent collaboration, external knowledge utilization, and AI integration within project management workflows. This paper concludes by discussing the future directions in AI-driven software development.

KEYWORDS

Artificial Intelligence, Software Engineering, Multi-agent

1. INTRODUCTION

Since the invention of computers, software development has been closely tied to scientific computing and industrial applications. Early software systems were mainly used to handle complex mathematical computations and manage industrial processes. Software not only accelerated data processing and the construction of complex models but also greatly enhanced design precision and operational efficiency, playing a crucial role in driving technological progress and industrial automation.

With the popularization of personal computers, the application of software expanded into domestic and commercial realms. Operating systems such as Microsoft Windows and Linux, along with applications like Microsoft Office, email, and instant messaging tools, have significantly transformed daily life and improved work efficiency. They have made document processing, information exchange, and data management more convenient than ever before. Moreover, the development of websites and multimedia software has greatly facilitated communication and revolutionized entertainment methods. From video platforms like Netflix and YouTube to various online news and blog platforms, these software applications not only provide a diverse array of information and entertainment resources but also allow individuals easy access to global information, enjoying various media content, thereby further deepening global cultural exchange and understanding.

After entering the era of smartphones, the explosive growth of mobile software has profoundly impacted all aspects of society. From social media apps to online shopping platforms, and mobile games, these applications continuously reshape communication methods, shopping habits, and entertainment. For instance, apps like Facebook, Instagram, and TikTok have not only transformed the face of social interaction but have also driven the development of new marketing strategies. Meanwhile, service apps like Uber and Airbnb have completely altered the traditional models of transportation and tourism, demonstrating the irreplaceable role of software in modern society. These transformations indicate that software technology is increasingly becoming a significant force in the socio-economic structures and cultural forms of society.

It is foreseeable that with the development of virtual worlds, digital twins, and metaverse technologies, the potential for software development will expand significantly in the future. In virtual environments, engineers have applied iterative optimizations to physical models to design tables that can move autonomously [1]. Additionally, digital twin technology has demonstrated its unique value in industry; for example, building digital twin models of equipment to optimize production processes. As these technologies continue to evolve and integrate, the role of software extends beyond the realm of information technology; it has become a critical force driving a wide range of socio-economic activities. Therefore, the future development of software will be a dynamic process filled with innovation. It is certain that these technological advancements will bring significant opportunities and challenges to software engineering.

With the growing complexity of technological systems, software complexity also continues to increase. For instance, the 2001 release of Red Hat Linux 7.1 contained 30 million lines of source code, of which the Linux kernel itself comprised 2.4 million lines [2]. In small to medium projects, it is quite common to see tens of thousands to hundreds of thousands of lines of code, which often represent months or even years of collaborative effort by team members. Since the inception of software development, software engineering has evolved into a comprehensive discipline, including areas such as requirements analysis, system design, programming implementation, software testing, maintenance, and project management. Despite the existence of various software development process models, such as the waterfall model, incremental model, spiral model, and agile development, a number of challenges and difficulties still persist in the software development process, including complexity of requirement management, scale and complexity of software architecture, code quality management, team collaboration and communication, accumulation of technical debt, deployment and testing, continuous documentation updates and maintenance, etc.

Over the past several decades, software development has primarily relied on human effort, that is, manually writing code line by line. Although integrated development environments (IDE) and tools (such as code completion, syntax checking, and performance profiling) have made great advancements, the core model, centered around manual labor, has remained unchanged. However, since the introduction of ChatGPT3.5 in 2022 [3], automated programming has become feasible and has quickly gained widespread attention and rapid development. Just as calculators and software once freed humans from the tedious task of numerical calculations (mathematicians no longer need to spend decades manually calculating π!), the introduction of AI represents a revolutionary change in the field of software engineering. This paper will explore the current directions in AI software engineering, address some of the core issues, and introduce some relevant applications.

2. RELATED BACKGROUND

The term "artificial intelligence" covers a broad range of technologies and application areas, such as pattern recognition, object detection and tracking, image classification, natural language processing, and machine translation. In this paper, we specifically refer to generative AI technologies based on large language models (LLMs). Taking GPT-3 as an example, it is an advanced large language model based on the generative pre-trained transformer (GPT) architecture [4]. The model employs a self-attention mechanism to handle input text, considering the full context of the input sequence to generate its output. ChatGPT is initially pre-trained on a huge dataset to learn broad patterns and structures of language, and then fine-tuned to perform more accurately in specific tasks. The primary advantage of this model is its capability to understand and generate human language, enabling it to fluently produce text close to human level.

LLMs have achieved remarkable success in natural language processing (NLP) tasks such as natural language conversation, text translation, and text summarization, drawing widespread attention. These models demonstrate powerful capabilities in context understanding, information extraction, and text generation. Since the rise of deep learning, researchers have explored its application to code generation [5][6]. Early attempts primarily relied on models such as recurrent neural networks (RNN) and long short-term memory networks (LSTM), often combined with structured code representations such as abstract syntax trees (AST). However, constrained by the model's ability and the scale of training data, these studies lacked accuracy and robustness in code generation. The generated code frequently contained syntax errors, logical flaws, or failed to effectively address real-world problems.

The breakthrough success of the GPT-3.5 architecture marked a new phase for LLM technology. Its powerful language modeling abilities, coupled with zero-shot and few-shot learning capabilities, offered new opportunities in the field of code generation. Utilizing the large amount of available open-source code (e.g., GitHub, Stack Overflow, etc.) as training data, and exploiting the powerful transformer architecture, a few models with code generation capabilities have emerged, including Codex, CodeLlama, and CodeGen. These models have demonstrated promising performance in tasks such as code completion, code translation, and unit test generation, although their capabilities are still under development and require further refinement.

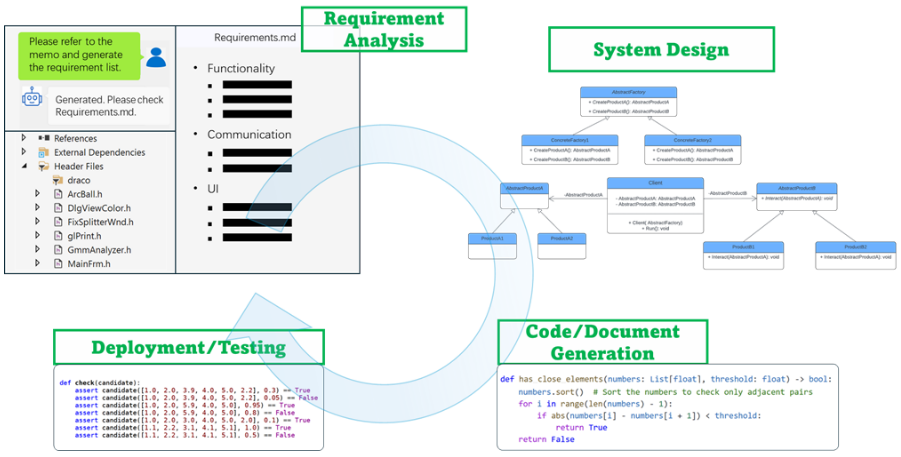
The rapid advancement of AI-powered code generation has led to the emergence of a range of commercial products. For instance, Text2SQL translates natural language into SQL code, offering features like code explanation and issue resolution, thus lowering the technical barrier to database operations [7]. Cursor focuses on enhancing project management and user interaction within the software development workflow [8]. By integrating AI assistance into the code editing environment, Cursor significantly facilitates the management, updating, and debugging of multiple code files, thereby enhancing developer productivity. Similarly, ChatGPT has introduced a canvas mode for code development, providing a more convenient interface that facilitates code modification and testing. Although these products and technologies are not yet capable of fully automating the development of large-scale software, they represent valuable contributions to AI-integrated software development.

3. AI INTEGRATION IN SOFTWARE DEVELOPMENT

First, let us begin by examining a common development process (as shown in Figure 1), exploring how AI can be effectively integrated into the software development workflow.

(1) **Generating a requirements list**. In software development, clearly defining the project's requirements is an essential first step that sets the objectives and expected functionalities. Typically, this process is carried out by human engineers through brainstorming, negotiation, and communication. In this process, AI technology can play the following roles:

- Utilizing natural language processing techniques to analyze customer-provided requirement documents, AI can automatically extract and organize functional requirements. For example, AI can identify key requirement elements from the documents, such as functionality, performance specifications, and user interface preferences, and convert this information into a structured requirements list.



Please refer to the memo and generate the requirement list.

Generated. Please check Requirements.md.

```
def has_close_elements(numbers: List[float], threshold: float) -> bool:
    numbers.sort()  # Sort the numbers to check only adjacent pairs
    for i in range(len(numbers) - 1):
        if abs(numbers[i] - numbers[i + 1]) < threshold:
            return True
    return False
```

```
def check(candidate):
    assert candidate([1.0, 2.0, 3.9, 4.0, 5.0, 2.2], 0.3) == True
    assert candidate([1.0, 2.0, 3.9, 4.0, 5.0, 2.2], 0.05) == False
    assert candidate([1.0, 2.0, 5.9, 4.0, 5.0], 0.95) == True
    assert candidate([1.0, 2.0, 5.9, 4.0, 5.0], 0.8) == False
    assert candidate([1.0, 2.0, 3.0, 4.0, 5.0, 2.0], 0.1) == True
    assert candidate([1.1, 2.2, 3.1, 4.1, 5.1], 1.0) == True
    assert candidate([1.1, 2.2, 3.1, 4.1, 5.1], 0.5) == False
```

Figure 1. Overview of AI-driven software development

- Identifying any ambiguous or vague statements in the requirements, prompting developers to clarify or refine further, thereby ensuring the precision and feasibility of the requirements.

(2) **System design**. In traditional development workflows, senior engineers are responsible for constructing the overall framework and design of the system based on the project requirements. This involves considering factors such as functionality, available resources, performance requirements, and future extensibility. In this process, AI technology can play the following roles:

- Automated design recommendations: AI may automatically generate preliminary design proposals based on requirements, offering a variety of architectural options for developers to choose from, thus accelerating the design decision-making process.
- Performance optimization: AI may identify potential bottlenecks based on workload analysis across different scenarios, analyze and predict the performance of the system under different designs, and facilitate overall performance optimization.
- Risk assessment and management: AI may find out potential design risks and contradictions, providing early warnings to ensure the reliability and safety of the design.

(3) **Module based code/document generation**. Once the overall framework is clarified, developers need to begin the actual implementation by writing program code and concurrently generating corresponding technical documentation. This step is crucial for transforming the design into functional and maintainable software. In this process, AI technology can play the following roles:

- Automated code generation: AI may automatically generate function-level or module-level code based on the system design and requirement specifications, greatly increasing development speed and avoiding common programming errors and typos.
- Intelligent code review: AI tools may perform real-time code reviews during the development process, detecting potential coding errors and performance issues to ensure code quality.
- Automated document generation: As the code is developed, AI may automatically generate and update technical documentation, such as API documents and user manuals, ensuring that the documentation is consistently updated in sync with the code implementation.

(4) **Deployment/testing**. After completing certain stages of software development, developers need to deploy and test the software in real or test environments. This is a detail-oriented and labor-demanding process, which ensures the software's stability and performance meet expectations in actual conditions. In this process, AI technology can play the following roles:

- Intelligent test case generation: AI may automatically generate test cases based on the behavior of the application, covering more usage scenarios, thereby enhancing the comprehensiveness and efficiency of testing.
- Automated deployment: AI may automate the deployment process, ensuring that the software is rapidly and consistently deployed across various environments, reducing potential errors caused by inconsistencies in deployment.
- Fault diagnostics: By automatically collecting and providing feedback on relevant information during system failures, AI may facilitate intelligent analysis, enabling developers to quickly identify and address errors.

In summary, during the requirements definition phase, AI can enhance the accuracy and completeness of requirement analyses. In the system design phase, AI not only boosts design efficiency but also optimizes design patterns. During the coding phase, AI facilitates the efficient and high-quality development of code and documentation. In the deployment and testing phase, AI improves the efficiency and quality of software testing and deployment, and assists in rapid error localization. Overall, integrating AI into software engineering allows for the effective optimization of tasks throughout the software development lifecycle. This integration is expected to generate considerable enhancements in both the development efficiency and the overall quality of the software product.

4. Key Challenges in AI-Driven Software Development

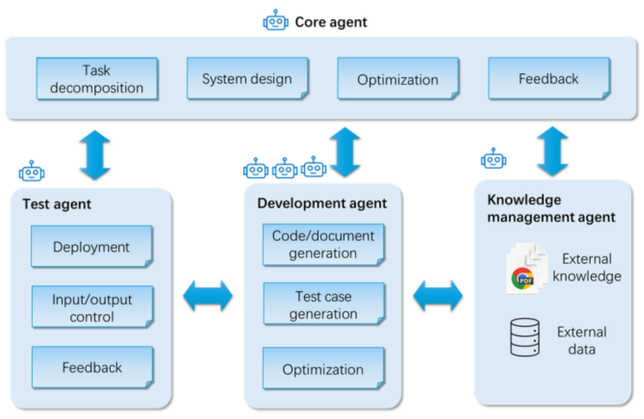


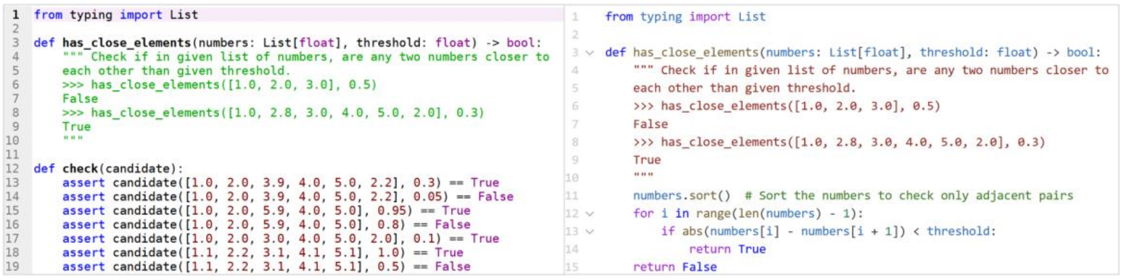
Figure 2. Workflow of AI agents in software development

The incorporation of AI in software engineering presents both opportunities and challenges. While there are expectations for significant improvements in productivity and software quality, substantial challenges also arise. In initial LLM applications such as translation or natural language conversation, a small degree of inaccuracy is acceptable, and users understand this limitation. However, programming demands extreme precision. Each line of code is translated into binary machine instructions executed by the CPU or GPU. Even within large-scale software systems, a minor but fatal error—such as a memory allocation fault or a formatting mistake—can lead to the collapse of the entire software system, or even the operating system itself. Such failures can have severe social impacts, as illustrated by the massive blue screen incident involving CrowdStrike in July 2024.

It is evident that current AI technology is not yet capable of fully automating the development of large-scale software projects. Imagine a scenario: developers instruct AI to create an operating system similar to Microsoft Windows, or to develop a document processing software within a Linux environment. Within hours or days, the AI autonomously generates the complete software. Following this, developers provide feedback and debugging information, utilized by the AI to continue refining and optimizing the software. After several iterations, the software reaches a release-ready state. While such a scenario is conceivable in the future, there is still a significant gap between today's AI capabilities and this objective. In this section, we will analyze and discuss key challenges on the research and development path toward this goal.

4.1. Development Agent: Module-Level Code Implementation

In code generation benchmarks such as HumanEval [9] and MBPP [10], function-level code implementation serves as the foundation for AI code generation. These datasets primarily focus on the correctness of individual functions, providing a baseline for evaluating the fundamental capabilities of AI-generated code.



```python
from typing import List

def has_close_elements(numbers: List[float], threshold: float) -> bool:
    """ Check if in given list of numbers, are any two numbers closer to
    each other than given threshold.
    >>> has_close_elements([1.0, 2.0, 3.0], 0.5)
    False
    >>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)
    True
    """


def check(candidate):
    assert candidate([1.0, 2.0, 3.9, 4.0, 5.0, 2.2], 0.3) == True
    assert candidate([1.0, 2.0, 3.9, 4.0, 5.0, 2.2], 0.05) == False
    assert candidate([1.0, 2.0, 5.9, 4.0, 5.0], 0.95) == True
    assert candidate([1.0, 2.0, 5.9, 4.0, 5.0], 0.8) == False
    assert candidate([1.0, 2.0, 3.0, 4.0, 5.0, 2.0], 0.1) == True
    assert candidate([1.1, 2.2, 3.1, 4.1, 5.1], 1.0) == True
    assert candidate([1.1, 2.2, 3.1, 4.1, 5.1], 0.5) == False
```

```python
from typing import List

def has_close_elements(numbers: List[float], threshold: float) -> bool:
    """ Check if in given list of numbers, are any two numbers closer to
    each other than given threshold.
    >>> has_close_elements([1.0, 2.0, 3.0], 0.5)
    False
    >>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)
    True
    """
    numbers.sort()  # Sort the numbers to check only adjacent pairs
    for i in range(len(numbers) - 1):
        if abs(numbers[i] - numbers[i + 1]) < threshold:
            return True
    return False
```

Figure3. An example from the HumanEval dataset (left) and the corresponding code generated by ChatGPT-4o (right)

The goal of a development agent extends beyond function-level code generation to encompass the complete development of single modules, comprising multiple related functions and data structures. The primary tasks include:

(1) **Functionality**: The generated code and document must fully implement all specified requirements of the module or function, leaving no missing features, based on the natural language description of the requirements.
(2) **Reliability and robustness**: Beyond generating syntactically correct code, the agent must ensure that the code operates correctly under various inputs and boundary conditions, adhering to the intended functionality and logic.

(3) **Conformance to specifications**: The behavior and output of the code must conform to the expected functional specifications and design documentation, including input/output formats, resource utilization methods, and error handling mechanisms.
(4) **Code optimization**: The agent should be able to optimize code by selecting appropriate data structures and algorithms to address performance bottlenecks, thereby improving runtime efficiency or reducing resource consumption.
(5) **Automated test case generation**: To validate the correctness and performance of the code, the agent needs to automatically generate comprehensive test cases, encompassing normal cases, boundary cases, and exceptional or error cases.

The performance of the development agent heavily relies on LLMs, making their fundamental capabilities crucial. As illustrated in Figure 3, natural language can convey general principles, but often lacks precision, leading to ambiguities. Conversely, specific examples, while providing precise guidance, typically cover only a limited number of cases. Consequently, AI needs to fully understand the requirements to provide appropriate solutions, rather than simply matching or applying known patterns. Based on existing LLMs, AI agents have provided promising results in handling function-level tasks. For instance, by incorporating GPT-4o with a language model debugger, Zhong et al. achieved Pass@1 code generation rate of 98.2% [11].

4.2. Core Agent: System-Level Code Implementation

In AI-driven software development, the core agent, responsible for task decomposition and agent scheduling, is crucial and represents a key technology in the entire development process. Its primary responsibilities include:

(1) **Complex task decomposition**

The core agent is expected to construct analytical pathways to break complex tasks into multiple subtasks. This is a highly challenging task. In GPT-o1, by simulating human cognitive process, the "chain of thought" functionality was introduced, demonstrating AI’s capability in task decomposition. In the research conducted by Zhu et al. [12], AI progressively learned rules during exploration in the Minecraft game world, and successfully decomposed the complex task of "obtaining diamond" into dependent subtasks.

Furthermore, after decomposing a complex task into a series of subtasks, the core agent must also validate the execution results of these subtasks to ensure their correctness and effectiveness. This implies that the core agent needs to design module tests for each independent subtask, as well as integration testing for the overall system.

(2) **System architecture design and optimization**

The core agent needs to promote modularity in system design, so that individual components can be assigned to development agents. In addition, it must be capable of continuously monitoring the system's operational status and rapidly diagnosing performance bottlenecks or failure points. This kind of system management helps in maintaining high reliability and availability. Based on performance analysis, the core agent should establish a feedback loop that allows for iterative improvements, thus optimizing system performance over time.

(3) **Multi-agent collaboration and scheduling**

The core agent, as the manager and coordinator of a multi-agent system, is responsible for constructing a collaborative framework and clearly defined workflows for agents. On this foundation, it is also essential to establish explicit protocols and interfaces to ensure efficient and consistent communication among different agents. In [13], a meta-programming framework named MetaGPT is introduced, aimed at simulating the software development process by assigning agents to different roles (such as product managers, architects, engineers, etc.) and setting standardized communication protocols (SOPs) to facilitate multi-agent collaboration.

4.3. Knowledge Management Agent

The core responsibility of the knowledge management agent is to manage externally provided knowledge bases and data, thereby offering essential knowledge support to other agents. This feature enables the software development process to meet specific user requirements, making it a critical component in the multi-agent systems.

Two primary approaches are commonly employed by knowledge management agents for handling external knowledge bases: retrieval augmented generation (RAG) and fine-tuning. RAG enables agents to augment their response generation process by first retrieving relevant information from external knowledge repositories. By combining information retrieval systems with generative models, RAG ensures that generated responses are more accurate, reliable, and reflective of current knowledge. A key advantage of RAG is its efficiency in knowledge updates; rather than requiring full model retraining, RAG dynamically incorporates new information through retrieval, thereby significantly reducing update and maintenance overhead. This makes RAG particularly suitable for dynamic knowledge domains where information changes frequently.

Fine-tuning, conversely, involves the further training of a pre-trained LLM using a specific knowledge base dataset. This process refines the model's understanding and application of domain-specific knowledge, leading to enhanced performance in targeted areas. While fine-tuning offers the benefit of deep knowledge integration, it demands considerable resources, including large amounts of labeled data and substantial computational power. A drawback is the need for complete retraining whenever the knowledge base is updated, making it less suitable for rapidly evolving knowledge domains.

Regarding external data, the knowledge management agent requires specific data manipulation capabilities to effectively manage and utilize information. These capabilities typically encompass the creation, deletion, modification, and retrieval of related data.

4.3. AI Integration in Software Project Management

Before the advent of integrated AI development environments like Cursor [8], even though AI possessed powerful capabilities, the only way for developers to collaborate with AI in software development was through the cumbersome process of copying and pasting code and debugging information between the chat dialog and local project files. This primitive method of interaction was not only highly inefficient and time-consuming but also severely limited developers' ability to handle multiple files simultaneously, leading to frequent context switches and low productivity. The integration of AI in software project management aims to break through these bottlenecks, significantly enhance development efficiency, and better address the various challenges throughout the software project lifecycle.

Within integrated AI development environments, to facilitate efficient collaboration and automation, AI needs to be granted the following permissions:

- Automated file/document interaction and updating: This encompasses the ability to programmatically access, modify, and manage files and documents within the project workspace.
- Document retrieval and access: This grants the AI agent the capacity to search, locate, and access relevant documentation, code comments, and other project-related information within the development environment.
- Automated input (keyboard and mouse control) and output information acquisition: This entails the ability to programmatically simulate user input via keyboard and mouse events, as well as capture and process output displayed on the screen or in the console. This enables the AI agent to automate tasks such as running tests, interacting with user interfaces, and collecting execution results.
- Acquisition of debugging information: This provides the AI agent with access to debugging information, such as logs, stack traces, variable values, and breakpoints. This allows the AI agent to assist developers in identifying and resolving errors, analyzing program behavior, and optimizing performance.

5. CONCLUSIONS

The rapid advancement of AI is profoundly reshaping the field of software engineering. The integration of AI with software engineering will undoubtedly greatly enhance development efficiency, and researchers alongside programmers are actively collaborating to push the boundaries of its capabilities. It is foreseeable that AI, trained on vast amounts of code, will become proficient in the various design patterns developed by human programmers. When addressing problems of low to medium complexity, AI is expected to demonstrate superior and more robust development capabilities than humans. However, when faced with highly complex systems and problems, the human programmer's abilities in analysis, abstraction, and creative problem-solving remain unmatched by AI at present. The critical challenge, and indeed the significant opportunity, lies in effectively combining human ingenuity with AI's computational power to revolutionize established software engineering methodologies. This exciting transformation may very well unfold within the next few years.

REFERENCES

[1] De Carpentier, G. (2024). Carpentopod: A walking table project. Retrieved from https://www.decarpentier.nl/carpentopod

[2] Linux (2024), In *Wikipedia, The Free Encyclopedia*. Retrieved from https://en.wikipedia.org/wiki/Linux

[3] OpenAI. (2022). ChatGPT — Release Notes. Retrieved from https://help.openai.com/en/articles/6825453-chatgpt-release-notes

[4] Brown, T., et al. (2020). Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, Vol. 33, pp. 1877-1901.

[5] Katsogiannis-Meimarakis, G., & Koutrika, G. (2023). A survey on deep learning approaches for text-to-SQL. *The VLDB Journal*, Vol. 32, pp. 905-936.

[6] Li, J., Wang, Y., Lyu, M. R., & King, I. (2017). Code completion with neural attention and pointer networks. *arXiv preprint arXiv:1711.09573.*

[7] Text2SQL.AI. (2024). Generate SQL with AI. Retrieved from https://www.text2sql.ai/

[8] Cursor. (2024). The AI Code Editor. Retrieved from https://www.cursor.com/

[9] Chen, M., et al. (2021). Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

[10] Austin, J., et al. (2021). Program Synthesis with Large Language Models. *arXiv preprint arXiv:2108.07732.*

[11] Zhong, L., Wang, Z., & Shang, J. (2024). Debug like a Human: A Large Language Model Debugger via Verifying Runtime Execution Step by Step. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 851-870.

[12] Zhu, X., et al. (2023). Ghost in the Minecraft: Generally capable agents for open-world environments via large language models with text-based knowledge and memory. *arXiv preprint arXiv:2305.17144.*

[13] Hong, S., et al. (2023). METAGPT: meta programming for a multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352.*

[14] Lewis, P., et al. (2020). Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, 33, pp. 9459-9474.

[15] Devlin, J., Chang, M., Lee, K., & Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.