From Today's Code to Tomorrow's Symphony: The AI Transformation of Developer's Routine by 2030

KETAI QIU, NICCOLÒ PUCCINELLI, and MATTEO CINISELLI, Università della Svizzera Italiana, Lugano, Switzerland
LUCA DI GRAZIA, Università della Svizzera Italiana, Lugano, Switzerland

In the rapidly evolving landscape of software engineering, the integration of AI into the Software Development Lifecycle (SDLC) heralds a transformative era for developers. Recently, we have assisted to a pivotal shift toward AI-assisted programming, exemplified by tools like GitHub Copilot and OpenAI's ChatGPT, which have become a crucial element for coding, debugging, and software design. In this article, we provide a comparative analysis between the current state of AI-assisted programming in 2024 and our projections for 2030, by exploring how AI advancements are set to enhance the implementation phase, fundamentally altering developers' roles from manual coders to orchestrators of AI-driven development ecosystems. We envision *HyperAssistant*, an augmented AI tool that offers comprehensive support to 2030 developers, addressing current limitations in mental health support, fault detection, code optimization, team interaction, and skill development. We emphasize AI as a complementary force, augmenting developers' capabilities rather than replacing them, leading to the creation of sophisticated, reliable, and secure software solutions. Our vision seeks to anticipate the evolution of programming practices, challenges, and future directions, shaping a new paradigm where developers and AI collaborate more closely, promising a significant leap in SE efficiency, security, and creativity.

CCS Concepts: • **Software and its engineering** → **Automatic programming**;

Additional Key Words and Phrases: Software Engineering, AI for Code, Human Factors in Software Engineering

ACM Reference format:
Ketai Qiu, Niccolò Puccinelli, Matteo Ciniselli, and Luca Di Grazia. 2025. From Today's Code to Tomorrow's Symphony: The AI Transformation of Developer's Routine by 2030. *ACM Trans. Softw. Eng. Methodol.* 34, 5, Article 121 (May 2025), 17 pages.
https://doi.org/10.1145/3709353

1 Introduction

Context. The evolution of software engineering and the integration of AI assistants, like GitHub Copilot [9] and ChatGPT [8], is dramatically changing daily routines of software developers [6, 27]. Several studies investigated the usage of these tools in the **Software Development Lifecycle**

This study was funded by Swiss National Science Foundation (SNSF) (Grant No, 215487).
Authors' Contact Information: Ketai Qiu, Università della Svizzera Italiana, Lugano, Switzerland; e-mail: ketai.qiu@usi.ch; Niccolò Puccinelli, Università della Svizzera Italiana, Lugano, Switzerland; e-mail: niccolo.puccinelli@usi.ch; Matteo Ciniselli, Università della Svizzera Italiana, Lugano, Switzerland; e-mail: matteo.ciniselli@usi.ch; Luca Di Grazia (corresponding author), Università della Svizzera Italiana, Lugano, Switzerland; e-mail: luca.di.grazia@usi.ch.

ACM 1557-7392/2025/5-ART121
https://doi.org/10.1145/3709353

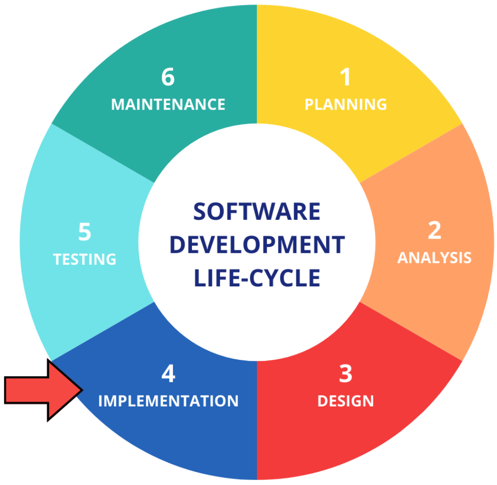


Fig. 1. The SDLC. We specifically focus on the implementation phase.

(SDLC) (Figure 1), evaluating how developers leverage them, showing the unprecedented support in coding, debugging, and even in the creative aspects of software design [33, 41, 47].

This change in the SDLC is evident across all stages, from planning to maintenance, and promises to enhance the overall software quality. For example, at the implementation stage, tools like GitHub Copilot and OpenAI's ChatGPT already offer real-time coding assistance and suggest optimizations, reducing the development time and improving code quality. AI advancements have also revolutionized the testing phase of the SDLC [16, 48]. AI can automatically generate test cases based on the requirements [35] and code [45], ensuring comprehensive coverage and identifying edge cases that might be overlooked by human testers. Finally, integrating AI into the SDLC provides benefits in the continuous learning mechanism. AI tools can learn from each project, continuously improving their suggestions and assistance, and consequently the overall software development.

Significance. The significance of leveraging AI in software engineering embodies substantial cost savings and innovation in a competitive market, by enhancing efficiency, reducing the incidence of bugs and accelerating the development time. These factors and the recent huge investments into this domain underscore the potential and the importance of AI in reshaping the future of software development [43]. Our vision is rooted in the belief that AI, much like a copilot in the cockpit, serves as an indispensable assistant rather than a replacement for human developers. By augmenting the capabilities of software engineers through AI, we anticipate a future where the synergy between human and AI will achieve the creation of sophisticated, reliable, and more secure software solutions.

Contributions. The importance of focusing on the implementation (coding) phase cannot be overstated. This phase is critical in determining the quality, reliability, and functionality of software products. Our article contributes to the ongoing discourse on the integration of AI in the SDLC [46] with a particular focus on this pivotal phase. Through a comparative analysis between current practices (2024) and future projections (2030), we aim to highlight the impact of AI tools on the implementation stage of the SDLC. This analysis will provide insights into how these advancements not only streamline processes but also fundamentally change the role of developers, shifting from manual coders to orchestrators of AI-driven development ecosystems. By examining the evolution of the implementation phase, we shape a future where developers and AI collaborate more closely, marking a significant leap forward in the field of SE. In summary, our main contributions are as follows:

— A comparative analysis to evaluate the impact of AI on the implementation phase of the SDLC, contrasting current practices in 2024 with projections for 2030.
— We discuss how advancements in AI not only streamline development processes but also significantly alter the role of developers, shifting them from manual coding to orchestrating AI-driven ecosystems.
— We envision a future where the integration of AI fosters a closer collaboration between developers and technology in the SDLC, representing a pivotal advancement in software engineering.

Article Structure. The article is organized as follows. Section 2 provides an historical introduction about programming, overviewing the limitations of the current approaches. Section 3 envisions how current limitations could be overcome; thanks to the introduction of *HyperAssistant*, an innovative augmented assistant that, by 2030, will possess the capability to provide comprehensive support to developers. Section 4 describes a hypothetical workday for a developer in 2024 and 2030, focusing on the limitations of the AI in 2024. Section 5 discusses the implications for developers and research in the upcoming years. Finally, Section 6 summarizes the content of the article and concludes the work.

2 Developers in 2024

The integration of AI with software engineering has unlocked new pathways for tackling a broad spectrum of challenges across various levels of abstraction. Concurrent with the advancements in hardware, particularly in GPUs [2], and the progression of techniques ranging from **Machine Learning (ML)** and deep learning [50] to **Large Language Models (LLMs)** [28], there has been a significant increase in efforts to leverage AI in software development. This synergy not only enhances the efficiency and effectiveness of software solutions but also paves the way for innovative approaches to complex problem-solving in software engineering.

2.1 Historical Evolution of Programming

Retrospecting the programming history, we can identify three distinct ages during the evolution of programming: pre-APIs, APIs, and LLMs, which witness the transition of software development from monolithic to microservice-based, and then to intellectual applications that involve enormous natural interactions between humans and software.

Before the advent of APIs, software applications were designed to operate in a standalone way, which means they were not able to interact with other systems. In other words, the interactions among different components of the system are tightly coupled with specific workflows during the era of pre-APIs. Although it is convenient for developers to build a monolithic application at the beginning of the software development, the consequential product will severely suffer from the lack of extensibility and flexibility.

Therefore, APIs are proposed to overcome limitations of monolithic applications [12]. APIs define a collection of protocols for the standardized communication between related systems. Since different systems are very likely implemented using different frameworks or programming languages, it is important to have a consistent and controllable way to share information with each other. Currently, Representational State Transfer and GraphQL[1] are the most frequently used

[1]https://graphql.org/

standards to design APIs for production usage. APIs are independent from the implementations of the interoperated systems, so they are flexible and scalable [7]. These characteristics also accelerate for software companies the transition of software development from monolithic to microservice-based architecture, where the interactions between distinct microservices are crucial [4].

Recently, LLMs are becoming a new manner of programming due to their incredible performance and convenience. LLMs are a special type of **Generative AI (GAI)** system that is capable of generating texts based on next token prediction. Hence, they're specially suitable for programming tasks with hundreds of lines of code. They're designed to be efficient, scalable, and flexible. These three unique advantages make LLMs highly popular for programming nowadays.

First, LLMs are efficient. With the help of LLMs, anyone can develop software applications even without domain knowledge by simply communicating with AI chatbots. End users can draft a method or a class in a few minutes by explaining, using the natural language, their requirements to LLMs, that are able to translate them into the implementation [25]. This communication process is officially called prompt engineering. From this perspective, it's essential to know how to efficiently convey specific requirements with LLMs. There are mainly two categories of prompts: zero-shot and few-shot, where "shot" refers to the simple explanatory example (input and expected output) within the relevant context provided by the users. With zero-shot learning, users do not provide any example to the model, relying on the former knowledge acquired by the model during the training, while in the few-shot learning, the model is provided with some clarifying examples.

Second, LLMs are scalable. Considering the ease of use of LLMs, it is essential to deploy LLMs on the cloud in a distributed way to serve requests from a handful of programmers. This can be achieved via distributing model shards across multiple GPUs [30] and caching prompts and the corresponding generated tokens as key–value pairs [31]. However, some developers prefer to deploy LLMs on local devices to secure confidential information during inference. Quantization is designed to facilitate this process by reducing the precision of the model's weights. Since LLMs usually have billions of parameters, users can save a significant amount of memory by using 8-bit floats or even 4-bit floats instead of 16-bit floats to store weights. There are several techniques (e.g., ZeroQuant [49] and GPTQ [18]) proposed to quantize LLMs while maintaining the performance.

Third, LLMs are flexible. Since LLMs are trained with a large volume of data collected from the Internet with a general purpose of inference, they may not perform well for a specific programming language. But they can be further tailored via fine-tuning or retrieval-augmented generation. For example, CodeLlama-70B-Python is built on top of Llama 2 for Python-related tasks,[2] which is an ideal tool for Python developers. Similarly, developers using other languages can easily adapt available LLMs for their specific programming tasks with few fine-tuning efforts.

In a nutshell, LLMs can not only guide junior programmers via tutorial conversations but also help senior developers accelerate the code understanding and the development process in an even faster agile manner, because they can delegate tedious coding parts to LLMs and mainly focus on the critical business logic [36].

2.2 Related Work

Back in 2012, Ammar et al. [1] surveyed the integration of AI techniques into software engineering processes, aiming to reduce development time and improve software quality. By seeking to bridge the gap between research and practice in applying AI to software engineering, they focused on requirements analysis, architecture design, coding, and testing, highlighting practical challenges and recent research in the area. Concurrently, Harman [22] directed attention toward a heightened level of abstraction, emphasizing the evolution within software engineering from conventional,

[2]https://ai.meta.com/blog/code-llama-large-language-model-coding/

localized, and clearly delineated construction methods toward the orchestration of expansive, interconnected, and intelligent systems. Harman's perspective focused on the several challenges ahead for AI integration in software engineering, such as the ways in which AI techniques can be used to gain insight to software engineers and the need for balancing automation with human intervention.

Three years later, Sorte et al. [46] provided a comprehensive overview of how AI techniques are integrated into various phases of the SDLC to automate and enhance the process. The authors explored the intersection of AI and software engineering, revealing that despite their separate development, these fields have much to offer each other. The article identifies key areas where AI contributes to software engineering (e.g., requirement specification, design, code generation, testing), while discussing relevant specific AI techniques for each phase of the SDLC. More recently, Shehab et al. [44] outlined the integration of AI into the SDLC, underscoring the potential of ML to enhance various phases of the software engineering process, including requirements engineering and code generation.

Nevertheless, in recent times, we have observed yet another paradigm shift with the emergence of GAI, which promises to significantly increase productivity [38] through the exploitation of natural language. Leading the charge are tools like OpenAI's ChatGPT, Github Copilot, and Google Gemini, which have become fundamental for developers owing to their remarkable capacity to augment productivity, foster creativity, and streamline efficiency [17, 41]. In order to understand how programmers interact with these system, Mozannar et al. [34] introduced the **CodeRec User Programming States (CUPS)** taxonomy, aimed at categorizing prevalent activities undertaken by programmers when utilizing these AI tools. The study involved 21 programmers who completed coding tasks and retrospectively labeled their sessions with CUPS categories. Key findings revealed significant time allocations toward activities tailored to interacting with Copilot. Notably, programmers frequently deferred suggestion verification, resulting in a notable portion of session time dedicated to managing Copilot's suggestions. These insights shed light on the inefficiencies and time overheads associated with the utilization of such systems.

In the realm of developer support, AI assistance has become a focal point of discussion. The rapid advancement of GAI in recent years prompts speculation on the extent of its future development and its potential to address researchers' concerns. Simultaneously, people are increasingly embracing and adapting these tools, as evidenced by the emergence of prompt engineering [14], which is gradually narrowing the divide between software and human interaction.

2.3 Limitations of the Current Approaches

The advent of generative AI as a developer's assistant has also introduced several challenges and opportunities among researchers and practitioners.

In the world of software development as in any other job, maintaining optimal mental health is crucial for sustained productivity, creativity, and overall happiness [21]. As developers navigate through intricate codebases and tight deadlines, the demands of the job can often take a toll on their mental and physical health. However, at this stage AI assistants focus mostly on solving technical challenges than human-factor problems related to coding activities.

Limitation 1: The mental health of programmers is frequently overlooked, despite its significant impact on productivity and overall well-being.

Despite the significant potential of AI in enhancing the software engineering process, especially with the recent advancements of automated code generation [39], over-reliance on such technology poses a risk in terms of security and code vulnerability. For example, the analysis of Pearce et al. [40]

revealed that approximately 40% of the generated programs with GitHub Copilot contains vulnerabilities and researchers are working on innovative solutions to avoid security issues introduced by AI-generated code [23]. Perry et al. [42] investigated whether developers relying on AI code assistants, like GitHub Copilot, produce less secure code than those who do not. By conducting a user study involving 47 participants, they found that those with access to an AI assistant produced significantly less secure solutions compared to those without access. On the other hand, the paper of Asare et al. [3] revealed that Copilot's likelihood of introducing vulnerabilities varies with the type of vulnerability and, most importantly, that GAI for code generation, while not perfect, does not perform worse than human developers in introducing vulnerabilities. These insights unquestionably establish a groundwork for further research in this domain.

Limitation 2: Developers often overestimate the capabilities of tools like GitHub Copilot by deferring the verification of the generated code, introducing more vulnerabilities and bugs.

Another relevant aspect to consider is the limits of these models in understanding semantic information. For example, Nie et al. [37] showed the significant improvement of the quality of the code generated for software testing when providing additional semantic information, like similar statements or the types of the variable defined, that can easily inferred by a developer but not by AI tools. Moreover, developers did not attribute the right importance to the code optimization, often using the copy and paste mechanism while programming. Several studies investigate this phenomenon [5, 24, 26, 29]. Kim et al. [26] showed that, despite the copy mostly involve single statements, snippets are copied in 25% of the cases, while Baker [5] reported the seek of performance as reason behind that, with developers that are evaluated on their productivity, pushing them to verbatim copy code rather than promoting refactoring of old code. Sometimes this behavior is unintention, with developers that tend to re-implement the same code since they are not aware of the presence of the same snippet [24].

This is reflected on the limited support for challenging tasks of the software development, like the optimization or the refactoring of the code.

Limitation 3: Modern tools frequently face challenges in code optimization, resulting in misleading errors and misunderstandings when developers fail to supply the necessary context.

Development's projects involve several developers, each of one contributing based on their own skills. For example, the developer that is most familiar with the database management is in charge of building a reliable data storing infrastructure, while the one with a lot of experience in the front end will develop an engaging Web page. Dividing these activities among different developers is far from trivial, and AI assistants are not able to improve the interaction between the team members, for example scheduling a meeting when a developer is struggling with the task at hand.

Limitation 4: AI tools are not helpful in promoting a synergical interaction between the team member, thus boosting the overall performance of the team.

Finally, the AI models are not fully integrated in the life of developers, and they are not able to take informed decision based on each developer, for example favoring new personalized learning paths. Programming languages are evolving over time. For example, in the last 30 years, more than 20 different Java versions have been released. Each version introduces new features (e.g., the *lambda expression* for Java 8) and makes obsolete some of them. AI tools are mostly trained on past data, and usually recommend popular solutions, lacking the capability to harness recent language advancements and present an innovative solution to the problem.

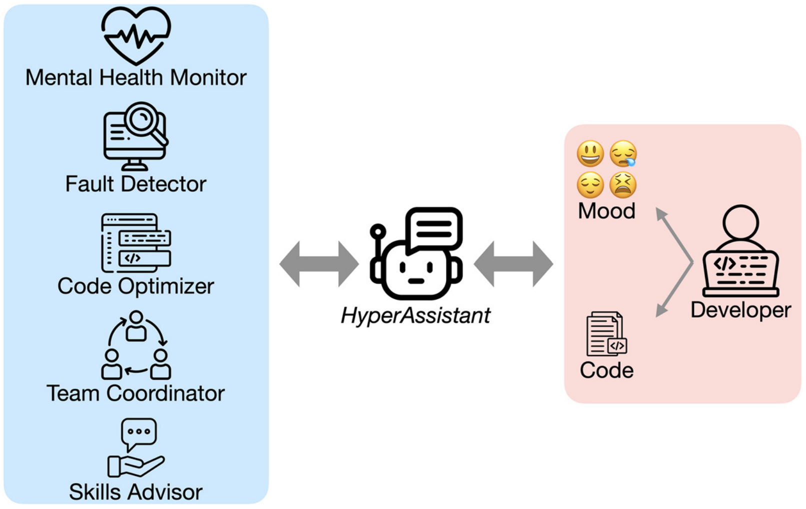


Fig. 2. Overview of *HyperAssistant* workflow and its components to improve developer productivity.

Limitation 5: Actual models fail to consider the unique needs and skills of programmers, lacking on personalized learning resources for software engineers.

3 Developers in 2030

In the previous section, we present the limitations of the novel tools in the software engineering tasks. Despite the good results achieved, they can still enhance the support provided to developers during the daily routine. In this section, we envision how developers may benefit from *HyperAssistant*, a hypothetical augmented assistant able to fully support developers in 2030. The complete structure of *HyperAssistant* is shown in Figure 2.

Our proposed system, i.e., *HyperAssistant*, is composed of five subsystems. The mental health monitor is specifically designed for tracking the mental state of developers. The fault detector is used to guarantee quality and reliability of the software under development. The code optimizer aims to accelerate the coding phase via automatic code completion and automatic code review. The team coordinator helps reduce invalid or repetitive communication as much as possible. The skills advisor facilitates continuous learning for developers. Each subsystem takes the developer's current mood and code into account and generates suggestions accordingly like a chat agent. The functionality and the underlying idea of each subsystem is described in detail as follows.

3.1 Mental Health

The integration of *HyperAssistant* could emerge as a transformative ally for mental health, offering innovative approaches to support and prioritize developers' well-being, as it is an essential topic for their productivity [20]. Here, we delve into three critical areas where *HyperAssistant* interventions can significantly impact mental health and the general well-being of software developers.

First, developers often find themselves immersed in coding sessions for many hours, leading to mental fatigue and decreased productivity. *HyperAssistant* can monitor developers' activity levels and cognitive performance in real-time, identifying signs of fatigue or stress. For example, *HyperAssistant* algorithms can analyze typing patterns, code quality metrics, and even biometric data to detect when a developer might benefit from a break. By suggesting timely breaks, such as recommending a short walk or a brief mindfulness exercise, *HyperAssistant* helps developers rejuvenate their minds and maintain optimal focus throughout their workday. As a result, imagine a scenario where a developer has been coding for several hours and begins to make frequent syntax errors or experiences difficulty in concentrating. AI, equipped with ML models trained on developers' behavioral patterns, recognizes these signs of cognitive fatigue and prompts the developer to take a 10-minute break. During this break, AI suggests engaging in breathing exercises or listening to calming music, fostering relaxation and mental clarity upon return to work.

Second, *HyperAssistant* can enhance **Integrated Development Environment (IDE)** environments by personalizing visual elements, such as color schemes and font sizes, to reduce eye strain and enhance readability. Additionally, AI-powered features can adjust ambient lighting and background music to create a more conducive atmosphere for concentration and positive emotions for developers [19]. As a result, consider a developer who spends long hours coding late into the night. *HyperAssistant*, aware of the time and the developer's preferences, adjusts the IDE's color scheme to reduce blue light exposure, promoting better sleep quality. Furthermore, based on the developer's music preferences and mood indicators, *HyperAssistant* selects instrumental tracks with a soothing tempo to create a calming ambiance conducive to focused work.

Last, developers' mental and physical health are influenced by various factors beyond their coding activities, including exercise routines, dietary habits, and personal stressors. *HyperAssistant* can analyze developers' lifestyle data, such as fitness tracker metrics and self-reported wellness indicators, to offer personalized suggestions tailored to their unique needs and preferences. As a result, suppose a developer has been experiencing increased stress levels due to a combination of work pressure and personal commitments. *HyperAssistant*, integrated with the developer's calendar and fitness tracker, recognizes patterns indicating high stress levels and suggests incorporating short exercise breaks or mindfulness practices into their daily routine. Additionally, *HyperAssistant* may recommend healthy meal options or provide tips for improving sleep hygiene, addressing holistic aspects of the developer's well-being beyond the confines of coding tasks.

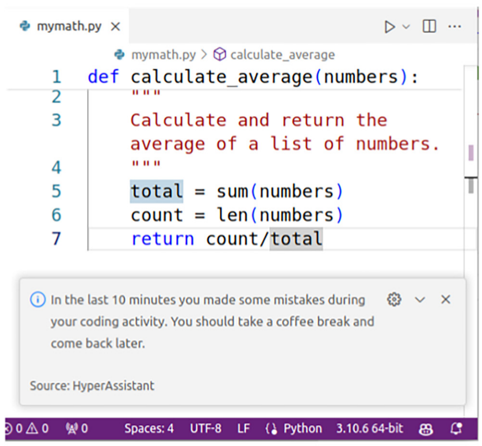
Through these examples, it becomes evident that AI-driven interventions have the potential to profoundly impact developers' mental health by providing proactive support, optimizing their work environment, and addressing broader lifestyle factors. Figure 3(a) shows an example of *HyperAssistant* for this task. By prioritizing mental well-being alongside technical proficiency, developers can foster a healthier and more sustainable approach to software development, ultimately leading to enhanced creativity, productivity, and job satisfaction.

Solution 1: *HyperAssistant* has the potential to profoundly impact developers' mental health by providing proactive support and optimizing their work environment.

3.2 Fault Detection

In the dynamic landscape of software development, the need for bug-free code remains essential [13]. *HyperAssistant* emerges as a transformative force, offering novel opportunities for bug detection and bug fixing. Here, we delve into critical tasks where *HyperAssistant* stands to revolutionize bug detection: vulnerability recognition, static analysis integration, and test case generation. Through these endeavors, *HyperAssistant* empowers developers to improve software quality and assist in delivering more resilient software solutions.

First, *HyperAssistant* can analyze the codebase for potential security vulnerabilities by recognizing patterns indicative of common security flaws. For instance, it can detect simple logic errors in real-time as Figure 3(b) shows or even complex errors such as instances where user input is not properly sanitized before being used in **Structured Query Language (SQL)** queries, potentially leading to SQL injection attacks. Upon detection, AI can suggest fixes or propose code modifications to mitigate these vulnerabilities, such as using parameterized queries instead of concatenating

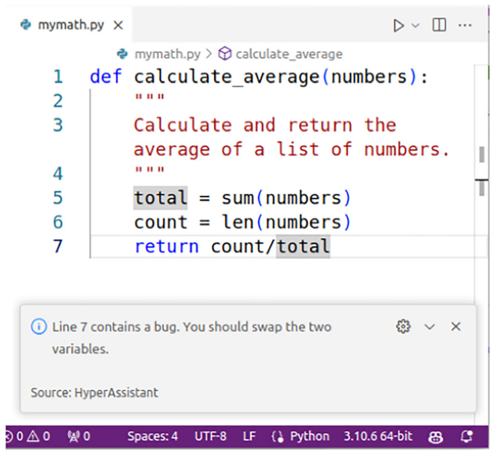

mymath.py
mymath.py > calculate_average
0 ⚠ 0 0 Spaces: 4 UTF-8 LF {} Python 3.10.6 64-bit

```python
def calculate_average(numbers):
    """
    Calculate and return the
    average of a list of numbers.
    """
    total = sum(numbers)
    count = len(numbers)
    return count/total
```

In the last 10 minutes you made some mistakes during your coding activity. You should take a coffee break and come back later.

Source: HyperAssistant

(a) AI suggests that the developer take a break after it detects signs of tiredness.

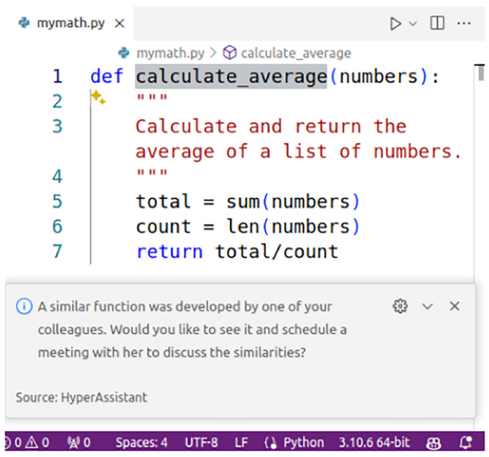

mymath.py
mymath.py > calculate_average
0 ⚠ 0 0 Spaces: 4 UTF-8 LF {} Python 3.10.6 64-bit

```python
1  def calculate_average(numbers):
2      """
3      Calculate and return the
       average of a list of numbers.
4      """
5      total = sum(numbers)
6      count = len(numbers)
7      return count/total
```

Line 7 contains a bug. You should swap the two variables.

Source: HyperAssistant

(b) AI suggests a bug fix while the developer is coding.

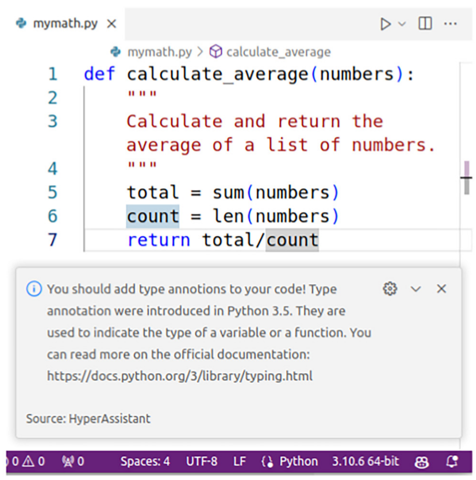

mymath.py
mymath.py > calculate_average
0 ⚠ 0 0 Spaces: 4 UTF-8 LF {} Python 3.10.6 64-bit

```python
def calculate_average(numbers):
    """
    Calculate and return the
    average of a list of numbers.
    """
    total = sum(numbers)
    count = len(numbers)
    return total/count
```

A similar function was developed by one of your colleagues. Would you like to see it and schedule a meeting with her to discuss the similarities?

Source: HyperAssistant

(c) AI finds a similar method in the code base and suggests a team interaction.


mymath.py
mymath.py > calculate_average
0 ⚠ 0 0 Spaces: 4 UTF-8 LF {} Python 3.10.6 64-bit

```python
def calculate_average(numbers):
    """
    Calculate and return the
    average of a list of numbers.
    """
    total = sum(numbers)
    count = len(numbers)
    return total/count
```

You should add type annotions to your code! Type annotation were introduced in Python 3.5. They are used to indicate the type of a variable or a function. You can read more on the official documentation: https://docs.python.org/3/library/typing.html

Source: HyperAssistant

(d) AI suggests a new feature with related documentation.

Fig. 3. Various *HyperAssistant* suggestions for improving developer working routine in 2030.

strings for SQL statements. Moreover, AI can integrate with static analysis tools like type checkers or linters to perform comprehensive code analysis. For example, in Python development, AI can automatically run type checkers like MyPy to identify type inconsistencies or potential type-related errors in the codebase [10, 15]. By flagging such issues, developers can ensure code robustness and maintainability.

Second, upon completion of a function or a class, *HyperAssistant* can automatically execute it within a sandbox environment to assess its functionality. By running various test cases, *HyperAssistant* can simulate different input scenarios to detect any unexpected behavior or failures. For instance, if a function is supposed to sort an array, *HyperAssistant* can generate random arrays of varying sizes and contents to validate the sorting algorithm's correctness.

Last, *HyperAssistant* can assist in generating test cases automatically based on code coverage analysis and behavior prediction. By analyzing the code structure and potential execution paths, AI can generate test inputs that aim to maximize code coverage and uncover edge cases. This proactive approach to test case generation can help improve software quality by identifying bugs early in the development cycle.

Solution 2: *HyperAssistant* will offer advanced capabilities for bug detection and bug fixing even for complex software systems.

3.3 Code Optimization

Generating optimized code and reducing development times, for example reusing existing and efficient code, is crucial. In 2030, *HyperAssistant* will be able to monitor in real-time the code written by developers, being able to find whether the same code is present in the code base, thus preventing its duplication. Figure 3(c) shows an example. Moreover, *HyperAssistant* will also monitor online resources, like StackOverflow, and will suggest to the developer a specific snippet available online, dramatically reducing the development time.

This support can be extended also to related activities, like commenting and refactoring. Comments are a crucial component in software development, able to convey meaningful information and make the code clearer. However, sometimes they become obsolete, with developers that change the code without updating existing comments or javadoc [32]. This behavior is problematic and can have dangerous side effects, increasing the number of bugs in the code. *HyperAssistant* will help developers, ensuring consistency of comments with the written code, and raising warnings where issues are detected. *HyperAssistant* will also automatically update the comments, depending on the preference of each developer, even proposing a refactoring of the current solution and suggesting meaningful names for the variables.

Solution 3: *HyperAssistant* will be able to fully support the developers for code optimization, favoring the usage of existing code and the alignment between code and comments.

3.4 Smart Team Interactions

With the emergence of tools facilitating collaboration among numerous developers on the same project, like GitHub, ensuring an efficient interaction is essential.

HyperAssistant will boost this aspect, by favoring a more intense and synergical communication between developers, also improving the task assignment. *HyperAssistant* will be able to suggest a developer to ask the support of another team mate for writing a specific function, since it can realize that the code the developer is writing is more familiar or already written as Figure 3(c) shows. *HyperAssistant* will be fully integrated in the team, monitoring the activities of all the members and promoting smart interaction between them. It will be able to estimate the development time required for a specific task, based on the long observation of her programming activities, optimizing the overall coding pipeline.

Moreover, *HyperAssistant* will be used by the team to generate a first draft of the entire project: starting from the abstract design, *HyperAssistant* can write the entire code of the project and then can assign each part to a specific developer, the one who is more suitable based on her background knowledge, to check its correctness. *HyperAssistant* will also act like a reminder of certain tasks that are often forgotten by developers. It can, for example, monitor the activity of the developer and suggest to her that it is time to commit the results on GitHub, since it has completed the implementation of a specific function. It can also automatically schedule meetings between team members if it believes that this can be useful, in order to facilitate and improve the software development.

Solution 4: *HyperAssistant* will improve the interaction between developers, leading to an improved task distribution and translating abstract ideas into code.

3.5 Learning and Developing New Skills

Although programming languages evolve rapidly, developers' knowledge does not always keep the same pace, often remaining rooted in old yet reliable code. Ciniselli et al. [11] investigated the effect of the evolution of programming languages in the contest of AI tools, also showing that most of the methods extracted from GitHub belong to Java 8, released in 2014. This highlights an interesting trend for developers, that tend to stick to an old stable version rather than experimenting with novel features.

HyperAssistant can help in filling this gap, suggesting interesting articles to the developer, related to the code she is writing, and even recommending a new feature that may be useful in that situation. *HyperAssistant* can even propose learning courses or tutorials, helping the developer to fill skill gaps in specific areas and generating *ad hoc* tests for assessing the progress.

Solution 5: *HyperAssistant* will assist developers in honing their skills, proposing tailored learning resources promoting innovative features of the programming language.

4 Developer Daily Routine: 2024 vs. 2030

In this section, we envision a hypothetical working day for a developer in 2024 and in 2030. This description focuses on the limitations of the AI assistants in 2024, showing all the potential benefits of the novel technologies that will be developed in the future. In our example, *HyperAssistant* can seem intrusive, but each developer can define the information it can access to, thus personalizing their coding and life experience.

4.1 Developer Daily Routine in 2024

Ashley, the developer in 2024, arrives in the office at 8 a.m. in the morning. She immediately notices that the code she has written the day before has been changed by a colleague. Obviously, no comment at all! She spends 45 minutes trying to figure out the meaning of that code and finally she is ready to code. The task is quite demanding and she spends a couple of hours for the implementation of the new feature needed in this project. Clearly, a few typos in the code results in compilation errors, but Ashley patiently resolves them one by one. Finally it is compiling, but a few test cases give unexpected results. She spends several minutes asking GitHub Copilot to understand what was wrong with the code but the answers were too generic and useless so she decided to find the error by herself.

Stressed after 30 minutes of unsuccessful attempts at fixing that bug, she decides that it is time for a coffee break. In the coffee room, she meets Emma, a senior developer of her team, and she asks for help. Unfortunately, Emma is pretty busy so they can arrange a meeting for 2 p.m, right after lunch. Ashley has lunch lost in her thoughts, trying to understand what was the mistake. The meeting with Emma is extremely helpful. After several minutes of intense concentration, they realize that a javadoc Ashley has taken by correcting is outdated and not aligned with the code so she has to fix the code. Emma also suggests looking at a new API released in the last version of Java that is faster in case she needs to speed up the computation. Unfortunately, Emma has no time for further help since she has a really busy agenda. Ashley decides to start fixing the current version of the method since she is not aware of the new API and, after a stressful day, she does not want to spend time on a new learning task.

After 1 hour of effort, the code is working but, as Emma perceived, it is too slow. So she looks online to understand the new suggested API, but the resources are limited and unclear, and it requires more time than expected. Finally, the working day is over and now the code seems to work properly, even though an unfixable warning message leaves her pondering. Ashley can finally go home after a stressful and non-productive working day. And tomorrow (unfortunately) will be the same.

4.2 Developer Daily Routine in 2030

Ashley, the developer in 2030, arrives in the office and immediately notices that some code has changed since yesterday. However, thanks to *HyperAssistant*, a concise summary is presented to her, highlighting only the pertinent edits. With this efficiency, she swiftly comprehends the updates and is ready to begin her tasks.

As she starts coding, an intelligent bug detection system notifies her of an error she inadvertently introduced. The system not only reports the bug but also suggests potential fixes, streamlining the debugging process. Furthermore, Ashley receives a notification regarding misalignment between the code and its corresponding javadoc comments. *HyperAssistant* offers suggestions on how to align them properly, ensuring code clarity and documentation consistency.

During her work, *HyperAssistant* recommends a piece of code implemented by a senior developer in the same company, recognizing its relevance to Ashley's task. *HyperAssistant* schedules a meeting for them by accessing their calendar, providing Ashley with preparatory materials to review beforehand. Additionally, Ashley is encouraged to take a break before the meeting, as *HyperAssistant* detects signs of fatigue, such as typos or syntax errors during the last 10 minutes, due to a lack of sleep the previous night accessing data from her wearable.

Following the productive meeting, Ashley successfully incorporates the optimized code, enhancing the project's features. With the task completed, AI suggests a balanced lunch from the Company Restaurant Web site, considering Ashley's plans for an evening gym session to maintain her well-being.

What once constituted a full working day for a developer in 2024 is now efficiently accomplished in half a day, allowing Ashley to tackle more tasks with precision, collaboration, and self-care in mind.

5 Discussion and Future Work

The comparison between the working days of a developer in 2024 and 2030, summarized in Table 1, sheds light on the remarkable advancements in AI technologies and their impact on the developer's productivity. As a result, we discuss implications for developers and researchers in the community of software engineering.

5.1 Implications for Developers

In 2024, Ashley's day is characterized by manual efforts and limited support from technology. She encounters challenges such as a huge list of code changes made by colleagues, debugging errors independently, and struggling with outdated documentation. While she seeks assistance from a senior developer, Emma, the help is constrained by Emma's busy schedule. Ashley's reliance on online resources for learning new APIs further slows down her progress. Despite her efforts, Ashley faces a stressful and unproductive day.

Conversely, in 2030, Ashley's experience is drastically different due to advancements in AI assistance. *HyperAssistant* streamlines her workflow by providing concise summaries of code

Table 1. Comparison between Developer Working Routine in 2024 and 2030

AI Assistance	**AI Limitations in 2024**	**AI Solutions in 2030**
Mental health	AI is not able to improve developers' mental health	AI can suggest the right moment for breaks and personalized activities to improve their well-being
Fault detection	AI is limited in automatically finding bugs and providing bug fixes for complex software systems	AI can automatically detect bugs and vulnerabilities, even recommending how to handle them
Code optimization	AI can recommend only simple code suggestions	AI can optimize the code, monitoring coding in real-time and suggesting alternatives
Smart team interactions	AI is not able to handle or suggest interactions between colleagues or teams in the same company	AI is able to support developers by arranging useful meetings and fostering developers' interactions
Learning new skills	AI cannot suggest relevant resources or novel programming language features or APIs	AI can find alternatives involving new features, proposing tailored learning paths for developers

changes and intelligent bug detection, significantly reducing the time spent on understanding modifications and debugging. The system also offers proactive suggestions for aligning code with documentation, enhancing code clarity and consistency. Furthermore, *HyperAssistant* facilitates collaboration by recommending relevant code implementations from within the company and scheduling meetings with colleagues, enabling efficient knowledge sharing and problem-solving. Moreover, *HyperAssistant* demonstrates a personalized approach by considering Ashley's well-being, detecting signs of fatigue, and recommending breaks and balanced meals. By leveraging data from wearable and company resources, *HyperAssistant* optimizes Ashley's productivity and supports her overall health.

Overall, the comparison highlights the transformative impact of AI technologies on developer workflows in 2030. Developers like Ashley can accomplish tasks more efficiently, collaborate effectively, and prioritize self-care, leading to increased productivity and job satisfaction. The limitations and challenges faced by developers in 2024 underscore the significance of advancements in AI-driven assistance, illustrating the potential benefits of embracing novel technologies in the workplace.

5.2 Implications for Researchers

The comparison between developer working routines in 2024 and those projected for 2030, as detailed in Table 1, not only underscores the transformative impact of AI on software engineering but also delineates crucial areas for future research.

Foremost, the evolution of AI in enhancing developers' mental health—from its initial inadequacy to a future where it intuitively recommends breaks and well-being activities—necessitates research into AI systems that can intricately understand and respond to individual health indicators. Practical exploration in this domain could involve collaborations between psychologists and software engineers to develop AI models that integrate psychological insights with real-time data analysis for personalized mental health recommendations.

The leap from basic fault detection to advanced, automated bug identification and resolution by 2030 invites the development of complex algorithms capable of deep code analysis. A practical research direction could involve partnerships with industry stakeholders to integrate these AI systems into existing development environments, enabling real-time, context-aware debugging suggestions based on historical project data and developer preferences.

In the sphere of code optimization, the shift toward AI that provides context-aware coding suggestions indicates a need for AI assistants that comprehend coding patterns, project intricacies, and optimization opportunities. Collaborative research with open-source communities could yield AI tools that learn from vast repositories of code to offer optimization advice, potentially even contributing code directly to projects.

The anticipated enhancement in smart team interactions through AI, transitioning from a non-existent to an active role in arranging and enhancing collaborations between developers, points to a future where AI aids in project management and team dynamics. Practical application of this research could see the creation of AI-driven platforms that analyze team performance data and project timelines to suggest optimal collaboration models and work distributions.

Lastly, the transition from AI's limited capability in suggesting new learning resources to a bespoke learning ecosystem tailored to developers' needs underscores the significance of AI in continuous professional development. This direction could be practically pursued by establishing partnerships with educational institutions and online learning platforms, utilizing AI to create dynamic, personalized learning pathways that adapt to the evolving technological landscape and individual learner goals.

These outlined paths pave the way for a future where AI not only boosts developer productivity but also plays a pivotal role in their professional and personal growth. Engaging in these research endeavors is crucial for unlocking AI's full potential in software engineering by 2030.

5.3 Technical and Research Challenges

The significant gap between the 2024 tools' capabilities and our vision for 2030 emphasizes some challenges that need to be addressed. Today's models lack in generating customized code that is adapted to a specific developer. They are not able to retain a significant amount of information about each developer, thus preventing the generation of suggestions that are familiar and adapted to their personal skills. This poses some challenges also for what concerns the improvement of the interactions between different developers, with models that may struggle in promoting synergical interactions based on the capabilities of each team member. To handle this challenge, research may focus on new and improved ways for creating contextual information that the model AI models can retain. The enhanced contextual knowledge can be leveraged to create a profile for each developer, thus enabling personalized suggestions and favoring team interactions.

Another limitation lies in the optimization of the generated code. AI assistants learn from the code used during the training, that can sometimes be low-quality, containing bugs or vulnerabilities. Hence, they also tend to recommend sub-optimal suggestions that can introduce serious vulnerability issues. A viable solution to mitigate this problem is the improvement of models' reasoning abilities, allowing them to infer possible drawbacks of the proposed solution and explore the effectiveness of alternative approaches.

6 Conclusion

In this article, we discuss how AI for software engineering can bridge the gap between the current limitations and the future potential in areas such as mental health support, fault detection, code optimization, team interaction, and learning new skills. The transition from a reactive to a proactive AI approach in software engineering underscores the technology's capability to not only understand and adapt to the individual needs of developers but also to foster a collaborative, efficient, and health-conscious working environment.

Furthermore, we discuss how AI can serve as a catalyst for interdisciplinary research, merging insights from psychology, education, and project management to create a holistic support system for developers. This collaborative approach is crucial for developing AI systems that are not only technically proficient but also attuned to the human aspects of software development.

In sum, as we discuss the capabilities of AI, it is clear that its integration into software development heralds a new era of efficiency and well-being. The continued exploration of AI's potential will undoubtedly lead to significant advancements, making the profession more fulfilling and productive.

References

[1] Hany Ammar, Walid Abdelmoez, and Mohamed Hamdi. 2012. Software engineering using artificial intelligence techniques: Current state and open problems. In *Proceedings of the First Taibah University International Conference on Computing and Information Technology (ICCIT 2012),* Vol. 52.

[2] Li Minn Ang and Kah Phooi Seng. 2021. GPU-based embedded intelligence architectures and applications. *Electronics* 10, 8 (2021), 952. DOI: https://doi.org/10.3390/electronics10080952

[3] Owura Asare, Meiyappan Nagappan, and N. Asokan. 2023. Is GitHub'S Copilot as bad as humans at introducing vulnerabilities in code? *Empirical Software Engineering* 28, 6 (Sep. 2023), 24 pages. DOI: https://doi.org/10.1007/s10664-023-10380-1

[4] Florian Auer, Valentina Lenarduzzi, Michael Felderer, and Davide Taibi. 2021. From monolithic systems to microservices: An assessment framework. *Information and Software Technology* 137 (2021), 106600. DOI: https://doi.org/10.1016/j.infsof.2021.106600

[5] Brenda S. Baker. 1995. On finding duplication and near-duplication in large software systems. In *Proceedings of 2nd Working Conference on Reverse Engineering.* IEEE, 86–95.

[6] Marco Barenkamp, Jonas Rebstadt, and Oliver Thomas. 2020. Applications of AI in classical software engineering. *AI Perspectives* 2 (Jul. 2020). DOI: https://doi.org/10.1186/s42467-020-00005-4

[7] Grzegorz Blinowski, Anna Ojdowska, and Adam Przybyłek. 2022. Monolithic vs. microservice architecture: A performance and scalability evaluation. *IEEE Access* 10 (2022), 20357–20374. DOI: https://doi.org/10.1109/ACCESS.2022.3152803

[8] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D. Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems.* H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (Eds.), Vol. 33, Curran Associates, Inc., 1877–1901. Retrieved from https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

[9] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. arXiv:2107.03374. Retrieved from https://arxiv.org/abs/2107.03374

[10] Yiu Wai Chow, Luca Di Grazia, and Michael Pradel. 2024. PyTy: Repairing static type errors in Python. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE).* IEEE Computer Society, 1–13.

[11] Matteo Ciniselli, Alberto Martin-Lopez, and Gabriele Bavota. 2024. On the generalizability of deep learning-based code completion across programming language versions. arXiv:2403.15149. Retrieved from https://arxiv.org/abs/2403.15149

[12] Cleidson R. B. de Souza, David Redmiles, Li-Te Cheng, David Millen, and John Patterson. 2004. How a good software practice thwarts collaboration: The multiple roles of APIs in software development. *ACM SIGSOFT Software Engineering Notes* 29, 6 (Oct. 2004), 221–230. DOI: https://doi.org/10.1145/1041685.1029925

[13] Giovanni Denaro, Noura El Moussa, Rahim Heydarov, Francesco Lomio, Mauro Pezzè, and Ketai Qiu. 2024. Predicting failures of autoscaling distributed applications. *Proceedings of the ACM on Software Engineering* 1, FSE, Article 87 (Jul. 2024), 22 pages. DOI: https://doi.org/10.1145/3660794

[14] Paul Denny, Viraj Kumar, and Nasser Giacaman. 2023. Conversing with Copilot: Exploring prompt engineering for solving CS1 problems using natural language. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE '23).* ACM, New York, NY, 1136–1142. DOI: https://doi.org/10.1145/3545945.3569823

[15] Luca Di Grazia and Michael Pradel. 2022. The evolution of type annotations in Python: An empirical study. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22).* ACM, New York, NY, 209–220. DOI: https://doi.org/10.1145/3540250.3549114

[16] Thomas Dohmke, Marco Iansiti, and Greg Richards. 2023. Sea change in software development: Economic and productivity analysis of the AI-powered developer lifecycle. arXiv:2306.15033. Retrieved from https://arxiv.org/abs/2306.15033

[17] Christof Ebert and Panos Louridas. 2023. Generative AI for software practitioners. *IEEE Software* 40 (Jul. 2023), 30–38. DOI: https://doi.org/10.1109/MS.2023.3265877

[18] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. 2022. GPTQ: Accurate post-training quantization for generative pre-trained transformers. arXiv:2210.17323. DOI: https://doi.org/10.48550/ARXIV.2210.17323

[19] Daniela Girardi, Filippo Lanubile, Nicole Novielli, and Alexander Serebrenik. 2022. Emotions and perceived productivity of software developers at the workplace. *IEEE Transactions on Software Engineering* 48, 9 (2022), 3326–3341. DOI: https://doi.org/10.1109/TSE.2021.3087906

[20] Daniel Graziotin, Fabian Fagerholm, Xiaofeng Wang, and Pekka Abrahamsson. 2017. Unhappy developers: Bad for themselves, bad for process, and bad for software product. In *Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C '17)*. IEEE, 362–364.

[21] Daniel Graziotin, Fabian Fagerholm, Xiaofeng Wang, and Pekka Abrahamsson. 2018. What happens when software developers are (un)happy. *Journal of Systems and Software* 140 (2018), 32–47.

[22] Mark Harman. 2012. The role of artificial intelligence in software engineering. In *Proceedings of the 2012 1st International Workshop on Realizing AI Synergies in Software Engineering (RAISE '12)*, 1–6. DOI: https://doi.org/10.1109/RAISE.2012.6227961

[23] Jingxuan He and Martin Vechev. 2023. Large language models for code: Security hardening and adversarial testing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23)*. ACM, New York, NY, 1865–1879. DOI: https://doi.org/10.1145/3576915.3623175

[24] Cory J. Kapser and Michael W. Godfrey. 2008. "Cloning considered harmful" considered harmful: Patterns of cloning in software. *Empirical Software Engineering* 13, 6 (2008), 645–692.

[25] Majeed Kazemitabaar, Xinying Hou, Austin Henley, Barbara Jane Ericson, David Weintrop, and Tovi Grossman. 2023. How novices use LLM-based code generators to solve CS1 coding tasks in a self-paced learning environment. In *Proceedings of the 23rd Koli Calling International Conference on Computing Education Research*, 1–12.

[26] Miryung Kim, Lawrence Bergman, Tessa Lau, and David Notkin. 2004. An ethnographic study of copy and paste programming practices in OOPL. In *Proceedings of the 2004 International Symposium on Empirical Software Engineering (ISESE '04)*. IEEE, 83–92.

[27] Milan Latinovic and Viktoria Pammer-Schindler. 2021. Automation and artificial intelligence in software engineering: Experiences, challenges, and opportunities. In *The 54th Hawaii International Conference on System Sciences*, 146–155. DOI: https://doi.org/10.24251/HICSS.2021.017

[28] Jiayin Li. 2024. The evolution, applications, and future prospects of large language models: An in-depth overview. *Applied and Computational Engineering* 35 (Jan. 2024), 234–244. DOI: https://doi.org/10.54254/2755-2721/35/20230399

[29] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. 2006. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering* 32, 3 (2006), 176–192.

[30] Xihui Lin, Yunan Zhang, Suyu Ge, Barun Patra, Vishrav Chaudhary, Hao Peng, and Xia Song. 2024. Efficient LLM training and serving with heterogeneous context sharding among attention heads. arXiv:2407.17678. Retrieved from https://arxiv.org/abs/2407.17678

[31] Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhuo Xu, Anastasios Kyrillidis, and Anshumali Shrivastava. 2023. Scissorhands: Exploiting the persistence of importance hypothesis for LLM KV cache compression at test time. In *Advances in Neural Information Processing Systems*. A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (Eds.), Vol. 36, Curran Associates, Inc., 52342–52364. Retrieved from https://proceedings.neurips.cc/paper_files/paper/2023/file/a452a7c6c463e4ae8fbdc614c6e983e6-Paper-Conference.pdf

[32] Zhongxin Liu, Xin Xia, David Lo, Meng Yan, and Shanping Li. 2021. Just-in-time obsolete comment detection and update. *IEEE Transactions on Software Engineering* 49, 1 (2021), 1–23.

[33] Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais, and Zhen Ming (Jack) Jiang. 2023. GitHub Copilot AI pair programmer: Asset or liability? *Journal of Systems and Software* 203 (2023), 111734. DOI: https://doi.org/10.1016/j.jss.2023.111734

[34] Hussein Mozannar, Gagan Bansal, Adam Fourney, and Eric Horvitz. 2022. Reading between the lines: Modeling user behavior and costs in AI-assisted programming. arXiv:2210.14306. Retrieved from https://api.semanticscholar.org/CorpusID:253117056

[35] Ahmad Mustafa, Wan Mohd Nasir Wan Kadir, Noraini Ibrahim, Muhammad Arif Shah, Muhammad Younas, Atif Khan, Mahdi Zareei, Faisal Alanazi, and Muhammad Arif. 2021. Automated test case generation from requirements: A systematic literature review. *Computers, Materials and Continua* 67 (Feb. 2021), 1819–1833. DOI: https://doi.org/10.32604/cmc.2021.014391

[36] D. Nam, A. Macvean, V. Hellendoorn, B. Vasilescu, and B. Myers. 2024. Using an LLM to help with code understanding. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. IEEE Computer Society, Los Alamitos, CA, 881–881. Retrieved from https://doi.ieeecomputersociety.org/

[37] Pengyu Nie, Rahul Banerjee, Junyi Jessy Li, Raymond J. Mooney, and Milos Gligoric. 2023. Learning deep semantics for test completion. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE '23)*. IEEE, 2111–2123.

[38] Shakked Noy and Whitney Zhang. 2023. Experimental evidence on the productivity effects of generative artificial intelligence. *Science* 381, 6654 (2023), 187–192. DOI: https://doi.org/10.1126/science.adh2586 https://www.science.org/doi/pdf/10.1126/science.adh2586

[39] Ayman Odeh, Nada Odeh, and Abdul Mohammed. 2024. A comparative review of AI techniques for automated code generation in software development: Advancements, challenges, and future directions. *TEM Journal* (Feb. 2024), 726–739. DOI: https://doi.org/10.18421/TEM131-76

[40] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions. In *Proceedings of the 43rd IEEE Symposium on Security and Privacy (SP '22)*. Institute of Electrical and Electronics Engineers Inc., 754–768. DOI: https://doi.org/10.1109/SP46214.2022.9833571

[41] Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. 2023. The impact of AI on developer productivity: Evidence from GitHub Copilot. arXiv:2302.06590. Retrieved from https://arxiv.org/abs/2302.06590

[42] Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. 2023. Do users write more insecure code with AI assistants?. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23)*. ACM, New York, NY, 2785–2799. DOI: https://doi.org/10.1145/3576915.3623157

[43] Francisco Ribeiro, José Nuno Castro de Macedo, Kanae Tsushima, Rui Abreu, and João Saraiva. 2023. GPT-3-powered type error debugging: Investigating the use of large language models for code repair. In *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering (SLE '23)*. ACM, New York, NY, 111–124. DOI: https://doi.org/10.1145/3623476.3623522

[44] Mohammad Shehab, Laith Abualigah, Muath Jarrah, Osama Alomari, and Mohammad Daoud. 2020. Artificial intelligence in software engineering and inverse: Review. *International Journal of Computer Integrated Manufacturing* 33 (Jun. 2020), 1–16. DOI: https://doi.org/10.1080/0951192X.2020.1780320

[45] Jiho Shin, Sepehr Hashtroudi, Hadi Hemmati, and Song Wang. 2024. Domain adaptation for deep unit test case generation. arXiv:2308.08033. Retrieved from https://arxiv.org/abs/2308.08033

[46] Bhagyashree Sorte, Pooja Joshi, and Vandana Jagtap. 2015. Use of artificial intelligence in software development life cycle: A state of the art review. *International Journal of Technology Management* 03 (Apr. 2015), 2309–4893.

[47] Giriprasad Sridhara, Ranjani H. G., and Sourav Mazumdar. 2023. ChatGPT: A study on its utility for ubiquitous software engineering tasks. arXiv:2305.16837. Retrieved from https://arxiv.org/abs/2305.16837

[48] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering* (2024), 1–27. DOI: https://doi.org/10.1109/TSE.2024.3368208

[49] Zhewei Yao, Reza Yazdani Aminabadi, Minjia Zhang, Xiaoxia Wu, Conglong Li, and Yuxiong He. 2022. ZeroQuant: Efficient and affordable post-training quantization for large-scale transformers. In *Advances in Neural Information Processing Systems*. S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (Eds.), Vol. 35, Curran Associates, Inc., 27168–27183. Retrieved from https://proceedings.neurips.cc/paper_files/paper/2022/file/adf7fa39d65e2983d724ff7da57f00ac-Paper-Conference.pdf

[50] Caiming Zhang and Yang Lu. 2021. Study on artificial intelligence: The state of the art and future prospects. *Journal of Industrial Information Integration* 23 (2021), 100224. DOI: https://doi.org/10.1016/j.jii.2021.100224

Received 28 March 2024; revised 27 September 2024; accepted 15 November 2024