The Future of AI-Driven Software Engineering

VALERIO TERRAGNI, ANNIE VELLA, PARTHA ROOP, and KELLY BLINCOE, The University of Auckland, Auckland, New Zealand

A paradigm shift is underway in Software Engineering, with AI systems such as LLMs playing an increasingly important role in boosting software development productivity. This trend is anticipated to persist. In the next years, we expect a growing symbiotic partnership between human software developers and AI. The Software Engineering research community cannot afford to overlook this trend; we must address the key research challenges posed by the integration of AI into the software development process. In this article, we present our vision of the future of software development in an AI-driven world and explore the key challenges that our research community should address to realize this vision.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; **Designing software**; **Software design engineering**;

Additional Key Words and Phrases: Software Engineering, Artificial Intelligence, Machine Learning, Large Language Models, APIs, Libraries, Software Testing, Requirements Engineering

ACM Reference format:
Valerio Terragni, Annie Vella, Partha Roop, and Kelly Blincoe. 2025. The Future of AI-Driven Software Engineering. *ACM Trans. Softw. Eng. Methodol.* 34, 5, Article 120 (May 2025), 20 pages.
https://doi.org/10.1145/3715003

1 Introduction

In the dawn of computing (1940s), programmers wrote machine code, consisting of binary instructions to directly program computers' hardware. It was quickly understood that programming *needed a higher level of abstraction from the hardware* [7]. This allowed programmers to write code that is more readable, understandable, and portable across different hardware. From assembly language (a more human-readable representation of machine code) to scripting languages (e.g., Python and JavaScript), the past 70 years of programming languages and practices have witnessed a continuous pursuit of a higher level of abstraction [29]. This is to increase developers' efficiency and at the same time cope with the demand for increasingly complex software systems.

While the introduction of high-level programming languages has played a major role in allowing software developers to write concise and expressive code, a paradigm shift occurred in the early 2000s with the widespread use of **Application Programming Interfaces (*APIs*)** *and libraries*.

This work was supported by the Marsden Fund Council from Government funding, administered by the Royal Society Te Apārangi, New Zealand.
Authors' Contact Information: Valerio Terragni (corresponding author), The University of Auckland, Auckland, New Zealand; e-mail: v.terragni@auckland.ac.nz; Annie Vella, The University of Auckland, Auckland, New Zealand; e-mail: avel920@aucklanduni.ac.nz; Partha Roop, The University of Auckland, Auckland, New Zealand; e-mail: p.roop@auckland.ac.nz; Kelly Blincoe, The University of Auckland, Auckland, New Zealand; e-mail: k.blincoe@auckland.ac.nz.

ACM 1557-7392/2025/5-ART120
https://doi.org/10.1145/3715003

Before that, programmers had to write extensive amounts of code to perform even basic tasks. The shift toward using APIs and libraries had a profound impact on the efficiency and capabilities of software development [49, 117]. Programming can now be informally summarized as *chaining the inputs and outputs of API calls*, allowing an even higher level of abstraction.

The intuitive, informative, and concise nature of variable and API names is bringing our programs closer to resembling *human language*. Additionally, the ongoing evolution of higher level programming languages unmistakably demonstrates a trend toward making language constructs more closely aligned with human speech [29]. *Can this trend continue and eventually programming will reach the pinnacle of abstraction: natural language?* This is very unlikely. Human speech lacks the basic criteria of programming languages (e.g., lack of ambiguity). However, this does not mean that software engineers cannot be aided in writing programs by specifying their intent in natural languages. Developers have been using **StackOverflow.com (SO)** to search for solutions of programming tasks using natural language as queries. Indeed, SO and similar Q&A websites for developers [88] have become crucial tools to boost developer productivity [65, 73, 77, 85, 87, 91].

The recent rise of **Large Language Models (LLMs)** [116], especially following the global launch of GPT3.5, GPT4o, and more recently, o1, by OpenAI, has brought another revolution of programming, rapidly overshadowing platforms like SO [20]. While program synthesis from natural language queries has been a subject of research for many years [32], the performance of recent LLMs has shown results that were unthinkable just a few years ago [16, 23, 27, 35]. Now, developers no longer need to search on SO for code snippets; instead, they can directly ask GPT (or other LLMs), and even have conversational interactions to better understand and improve the generated code. Recently, SO removed statistics on its daily visit counts and officially addressed concerns about declining web site traffic in a blog post.[1] The post acknowledges the decline in visits and attributes the trend to the release of GPT-4. We are witnessing a paradigm shift in software development where software engineers use LLMs and other AI systems to boost their productivity [25, 75]. We can confidently say that LLMs, alongside high-level programming languages, libraries, and developer Q&A websites, have become essential tools for modern software development [25].

LLMs are here to stay. Indeed, their capabilities and performance in a wide range of software engineering tasks are set to improve in the future. This is due to the increasing availability of open source code for training purposes, alongside the ongoing efforts of the AI community to enhance the LLM performance. As such, over the next decade, we anticipate that software engineers will continue to use LLMs (or similar AI systems) in software development.

Our research community must acknowledge and address the opportunities and challenges that arise from the use of AI in software engineering. Concerns persist regarding the quality of AI-generated code [56], with notable issues regarding security and privacy [109].

Yet, there are numerous opportunities presented by the versatile capabilities of LLMs, especially when fine-tuned for specific tasks, code bases, or company practices. For example, recent research demonstrates that fine-tuned LLMs outperform general-purpose LLMs in code review tasks [68, 74]. Indeed, software engineering involves much more than writing code. LLMs have proven highly effective in various software engineering tasks beyond code generation [36], including documentation generation [30, 61], testing [79, 112], code review [68, 74], and program repair [45, 100, 101].

Our research community stands at the forefront of this revolution, we need to tempestively address the challenges of the *symbiotic partnership between human developers and AI*.

In this article, we present our vision of the potential future of an AI-driven software engineering, alongside the key research challenges and opportunities associated with the increasing integration

[1] https://stackoverflow.blog/2023/08/08/insights-into-stack-overflows-traffic/

of AI into the software engineering process. In particular, we propose the conceptual design of a framework to harness AI capabilities for automating, augmenting, and optimizing various stages of the software development lifecycle, including requirements analysis, design, coding, testing, and maintenance.

2 A Current Snapshot of AI Tools in Software Development

This section highlights the examples of current tools and platforms that demonstrate how AI (in particular LLMs) is transforming software engineering workflows. While this is not an exhaustive list, it includes some popular and well-established tools that are widely applicable in development pipelines. Research prototypes are excluded, focusing instead on mature tools ready for real-world use. These examples cover specific applications such as code generation, documentation, and bug fixing, showing how LLMs are becoming integral to modern software engineering practices. We acknowledge that this landscape is shifting very rapidly, and these tools, while the state-of-the-art at the time of writing in December 2024, could be superseded even by the time this article is published.

GitHub Copilot[2] is a pioneering tool in AI-powered development owned by Microsoft. It is powered by OpenAI's LLMs and can be integrated with popular **Integrated Development Environments (IDE)** to assist developers by suggesting code completions. It offers context-aware code completions based on the current files and project structure. The 2024 Q2 financial report of Microsoft reports that GitHub Copilot grew its paid customer base by 30% quarter-over-quarter to a total of 1.3 million developers and 50,000 organizations.[3] Tools like GitHub Copilot have been found to significantly boost software developer productivity [118].

Amazon released *CodeWhisperer*,[4] its own version of an AI code assistant. Different from GitHub Copilot, which is general purpose, CodeWhisperer specializes in AWS cloud development, providing tailored guidance and suggestions for AWS-specific coding and infrastructure.

Another recent example is *Windsurf by Codeium*,[5] recognized as the first "agentic IDE" that seamlessly incorporates AI features into development that goes beyond code suggestions. It is a fork of Visual Studio Code that enables developers to—(i) Prompt the AI to build an entire application, breaking it down into manageable tasks; (ii) Automatically create files, suggest package installations, and manage dependencies; and (iii) Review and refine its own code.

Similarly, *Cursor*,[6] another fork of Visual Studio Code, automatically fixes generated code and supports app creation by breaking tasks into smaller steps. Cursor offers extensive customization through the `.cursorrules` file format, enabling developers to adapt AI behavior for specific frameworks and languages.[7]

LLMs are also reshaping the way documentation is created and optimized. Tools such as *LLM Text*[8] help developers extract and summarize relevant context from various sources, including GitHub repositories, npm packages, and YouTube videos. *UiHub*[9] is another tool that leverages LLMs for software engineering. It enables large-scale repository analysis, allowing developers to build advanced development tools and applications.

[2] https://github.com/features/copilot
[3] https://www.microsoft.com/en-us/investor/events/fy-2024/earnings-fy-2024-q2
[4] https://docs.aws.amazon.com/codewhisperer/
[5] https://codeium.com/windsurf
[6] https://www.cursor.com/
[7] https://dotcursorrules.com/
[8] http://llmtext.com/
[9] https://uithub.com/

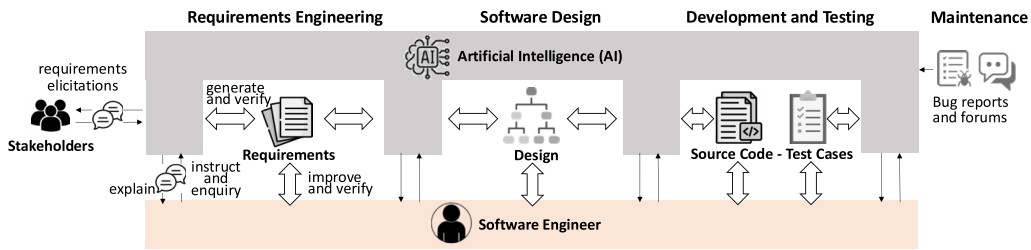

Fig. 1. Logical architecture of the envisioned future symbiosis of software engineers and AI.

These concrete examples showcase how current LLMs are effectively being integrated into software engineering, offering developers powerful capabilities to boost productivity, streamline workflows, and improve code quality. The few examples we discussed unmistakenly show that the evolution of software development is increasingly pointing toward *autonomous multi-agent systems*, where AI systems independently make decisions during software development [34].

This evolution inspires, motivates, and informs our framework for the future of software engineering, providing evidence that we are heading in that direction. As we embrace this trajectory, it becomes crucial to address the challenges ahead and necessitates rethinking how to better accommodate these intelligent AI-powered agent collaborators.

3 AI-Driven Software Engineering

Figure 1 overviews our envisioned *AI-driven software development framework*. While certain aspects of this framework may appear overly optimistic about the capabilities of future AI systems, it presents an interesting thought process for understanding the potential symbiotic synergy between AI and software developers. Moreover, it sheds light on the research challenges that our community must address to realize this vision someday. Indeed, such a vision is not completely unrealistic. We know that current AI systems can accomplish most of the specified tasks, albeit with limited quality [30, 38, 45, 61, 79, 100, 112].

The framework touches all main phases of the *Software Development Lifecycle*—requirement engineering, software design, implementation, testing, and maintenance. Note that we are not assuming a waterfall model, the cycles may overlap, especially in agile development methodologies where development cycles are shorter and more flexible.

The main Actors in our framework are *software engineers* (e.g., developers, architects, and testers) and a generic AI system (e.g., an LLM). It is important to mention that we believe we are still very far from completely replacing software engineers with prompt engineers. Capable software engineers (with prompt engineering training) will remain indispensable for understanding, reviewing, improving, combining, validating, and maintaining the source code generated by AI. In the short- and medium-term future, AI is merely a tool to enhance software developers' productivity. While it can automate certain tasks, we assume the presence of humans in the loop.

Stakeholders (e.g., end users, product owners) are also involved in the framework during the requirement engineering process. AI, equipped with chatbot capabilities, can initiate conversations with stakeholders to elicit, analyze, specify, and validate requirements.

With our proposed framework, software engineers can either directly create or update the artifacts (i.e., requirements, design, production, and test code) or instruct the AI (e.g., through prompt engineering) on how to do that. We envision a *bi-directional communication* between humans and AI, where humans can ask questions or provide instructions, and the AI can notify the software engineers of any detected issues or opportunities for improvement. Software engineers

will communicate with AI through *conversational interactions* facilitated by the conversational capabilities of LLMs. This interface empowers engineers to seek clarifications and explanations about the artifacts as well as the AI system's output [54].

Another important clarification is that, for simplicity, Figure 1 represents a single AI system. In reality, different tasks would be handled by specialized AI agents [99], each fine-tuned for its particular function. These subsystems may either operate autonomously or require direct input from software engineers, depending on the nature of the task.

In particular, the AI agents must effectively communicate with each other and with various software analysis tools responsible for gathering information on the software artifacts in development. As the number of available AI systems continues to grow, to prevent information overload, humans will interact with a single *unified interface*. Similar to mediator bots [76], an *orchestrator of AIs* can efficiently manage all interactions with the AI agents behind the scenes. We envision that the AI's orchestrator will constantly monitor changes in the artifacts (after every update from engineers or from the agents) and invoke the dedicated AI agent to check for consistency and integrity of the artifacts.

From a technical perspective, our framework is a multi-AI agent system [99]. AI agents are artificial entities that can autonomously perceive and act within their environment to achieve specific goals [99]. These agents typically consist of four key components: planning, memory, perception, and action [99]. The planning and memory components serve as the brain controlled by the AI (e.g., LLM), while perception and action enable the agent to interact with its environment and perform specific tasks. While a single-agent system is specialized for solving individual tasks, collaboration between multiple agents (i.e., multi-agent systems) enables the handling of more complex tasks. In the context of software engineering, multi-agent systems can support end-to-end software development processes [34, 55]. We envision that the AI orchestrator will coordinate the perception and action of each agent. We refer readers to the recent survey on LLM–based agents for software engineering by Liu et al. [55], which provides a comprehensive review of LLM-based agents in software engineering.

Achieving such a symbiotic partnership between human developers and AI presents several challenges. First, we must not underestimate the difficulties that the orchestrator will face in adapting LLM-based agents to fit existing software engineering tools, processes, and practices. It remains unclear whether entirely new processes and workflows are needed or if LLM-based agents can integrate seamlessly into current ones. Additionally, software developers need training to effectively use LLMs, including recognizing when they are hallucinating. While the framework will automate most interactions with the AI, we believe that communication between developers and AI will remain essential—especially when humans seek explanations for the AI's outputs or wish to provide creative insights to improve the software artifacts. To achieve this, developers must be proficient in prompt engineering to communicate effectively with LLMs and guide them toward desired outputs. This is a completely new skill for developers and should be incorporated into existing software engineering education curricula. Furthermore, while AI will undoubtedly be valuable in automating repetitive tasks, human creativity will continue to play a crucial role. Ensuring that creativity remains a dominant force in the workflow presents a significant challenge. Recognizing when AI requires human input versus when it can act independently will be critical.

> *AI Multi-Agent System*: The primary research challenge in integrating AI into the software development process will be orchestrating the various AI subsystems that focus on specific development tasks and seamlessly integrating them using a single human-AI interface.

3.1 Requirements Engineering

Gathering requirements accurately is crucial for the success of any software project [111]. Unclear or incomplete requirements can lead to misunderstandings, delays, and ultimately project failures. When requirements are not effectively gathered, developers will build features that do not align with stakeholder expectations and needs, resulting in wasted time and resources. However, this process is often challenging, as *understanding stakeholder needs* is a complex activity. For example, ambiguities in natural language, stakeholders not always knowing what they truly need or want, and evolving expectations contribute to this complexity. Moreover, translating complex functional or non-functional requirements into technical specifications can be difficult. This makes the requirements elicitation phase complex, time-consuming, and prone to errors.

Recent studies show that AI, particularly LLMs, can assist in requirements engineering activities [4, 59, 66]. These models are capable of analyzing, organizing, and summarizing large amounts of data, and, as a result, they can play a crucial role in the preliminary phase of requirements elicitation [66]. Stakeholders can provide documentation in any form, and LLMs can summarize large documents or translate them into formal requirement specifications. In particular, we envision AI supporting the following three key requirements engineering activities: elicitation, validation, and summarization.

AI-assisted Requirements Elicitation. AI-driven agents, such as chatbots powered by LLMs, could assist by engaging stakeholders in human-like conversations to elicit requirements. These agents are capable of generating clarifying questions and suggestions to help stakeholders articulate their needs more clearly. Moreover, they can propose relevant examples or scenarios to facilitate discussions and clarify ambiguities. For example, AI agents could produce mockups of interfaces or rapid prototypes to confirm understanding of user needs. This is particularly useful when stakeholders describe their envisioned solutions to a problem rather than the underlying problem. AI systems must ensure that stakeholders' proposed solutions do not limit innovative designs.

Dependency parsing can further enhance the elicitation process by structuring stakeholder input, identifying ambiguities, and generating follow-up questions to refine unclear requirements. It can also classify functional and non-functional requirements, ensuring that captured requirements are both structured and actionable [22].

Given the complexity and importance of requirements elicitation, we believe that software engineers will remain in the loop. They should oversee these AI-stakeholder interactions in real time or review them afterward, refining and validating the gathered requirements, and intervening when necessary to improve the elicitation process. While AI can facilitate communication, engineers are essential for interpreting non-verbal cues and ensuring stakeholders fully understand the discussion [97]. However, future advancements in *multimodal models* could allow AI systems to better interpret non-verbal cues such as facial expressions and gestures, enhancing their ability to fully capture stakeholder intent. This would reduce some of the reliance on engineers for this aspect, although human oversight would likely remain crucial for more complex, high-stakes discussions. This further emphasizes the need for an engineer's involvement in the elicitation phase, especially in cases where nuanced human communication is critical.

AI-Assisted Requirements Validation. While AI can contribute significantly during the elicitation phase, its true value lies in the validation of requirements. AI can continuously and automatically analyze requirements to detect inconsistencies, contradictions, and ambiguities. Using named entity recognition, AI can assist in refining these specifications by identifying vague references or conflicts in software requirements specifications, ensuring consistency and reducing ambiguity [64]. More importantly, AI can regularly monitor evolving software artifacts (e.g., source code, documentation, tests) to ensure alignment with the requirement and suggest corrective actions

when needed. Additionally, as AI systems increasingly participate in validation, it is important to ensure that their outputs are ethically sound and unbiased. Regular audits and transparency in how AI reaches decisions will help prevent biased recommendations from influencing projects, ensuring that validation processes remain fair and accurate. By maintaining a link between the requirements and software artifacts, AI could also predict the potential impact of changes in requirements by analyzing dependencies across the project, helping to assess risks of changing the requirements.

Requirement Summarization and Refinement. LLMs can assist by summarizing long or complex requirements, allowing engineers and stakeholders to focus on the most important details. By processing large volumes of text, LLMs can extract key points and streamline the language, making it easier to identify priorities. A challenge here will be to ensure that prioritization is fair and considers needs of minority groups (e.g., accessibility concerns). The natural language processing capabilities of AI can also help improve clarity by simplifying complex phrasing and ensuring that consistent terminology is used across all documents and discussions, reducing the risk of misunderstandings between teams.

> *Requirement Engineering*: The main research challenge will be to enable AI agents that can understand user needs to effectively validate the requirements ensuring accuracy and completeness.

3.2 Software Design

The integration of AI in software engineering holds immense potential in assisting with the design phase. Building on well-defined requirements, AI can assist in proposing initial design suggestions to align functional and non-functional needs. These suggestions can serve as starting points for further refinement and validation by the engineers and stakeholders. We believe that human involvement will remain essential in this step, as the design phase should not—and does not need to—be fully automated. Design artifacts are a vital communication tool between engineers and stakeholders. Additionally, AI can aid in design validation by generating visual artifacts, such as UML diagrams [1] or C4 models, helping engineers explore different solutions and ensure they meet the project's requirements. These design artifacts, ranging from prototypes to models and diagrams, offer a more structured and less ambiguous way to communicate design intentions compared to using natural language alone (requirements). We envision several key ways in which AI can enhance the design process and improve communication between engineers and stakeholders.

Prototypes Sketching. Requirements written in natural language, while essential for early-stage communication, often suffer from ambiguity, especially when describing complex systems. Prototypes and sketches offer a way to mitigate this issue by providing visual and structural representations of software design. AI-driven tools can automatically generate such artifacts based on requirements. These prototypes could be as simple as sketches, serving as a tangible starting point for discussions among engineers and stakeholders. This approach mirrors the use of paper sketches in traditional design processes, making early-stage design clearer and more accessible to non-technical audiences. This allows for early feedback and adjustments, accelerating the design iteration process. By creating these early-stage prototypes, AI not only aids in communication but also helps in validating ideas against requirements and constraints. AI can greatly improve this process by quickly and automatically generating mock UIs and tangible artifacts.

Multi-Level Design Artifacts for Different Stakeholders. One of the critical challenges in software design is communicating with stakeholders who have varying levels of technical expertise. While developers may require detailed, low-level design documents, project managers and business stakeholders often benefit from high-level, conceptual overviews. AI can facilitate the creation

of multi-level design artifacts that take into account these different audiences. For instance, AI could generate a high-level system overview using the C4 model,[10] which provides four layers of abstraction for system design—context, containers, components, and code. At the same time, the AI system could automatically produce more detailed, low-level diagrams for developers (e.g., UML class diagrams, UML sequence diagrams, state machine diagrams, component and deployment diagrams). By maintaining consistency across these different levels, AI will help ensure that all stakeholders, regardless of technical proficiency, have a clear and accurate understanding of the design. Moreover, AI can personalize the level of detail provided based on the user's role, dynamically adjusting the complexity of design artifacts. This adaptability enhances communication and collaboration between technical and non-technical teams, making software development more inclusive and efficient. We envision fine-tuned LLMs trained on design artifacts, source code, and requirements seamlessly generating and synchronizing these different levels of design artifacts.

Maintaining Up-to-Date Design Documentation. One of the persistent challenges in software development is keeping software design documentation synchronized with the actual codebase, especially as changes are made during development. AI could automate this process, ensuring that design artifacts reflect the current state of the system. For example, an AI system could be integrated with version control tools to automatically update architecture diagrams when code modifications are committed. This eliminates the need for manual diagram updates, which are often overlooked, leading to outdated and potentially misleading documentation. AI-driven automation of this process enhances the reliability of design documentation and ensures that it remains useful throughout the development lifecycle.

Validating Design Solutions. Beyond creating and maintaining design artifacts, AI can also contribute to the validation of design solutions. AI systems, trained on best practices and past project data, can analyze proposed designs for flaws or inefficiencies. For instance, AI could simulate different architectural choices and highlight potential bottlenecks, performance issues, or security vulnerabilities in early design stages. By using AI to validate designs, software engineers can receive real-time feedback on their decisions, reducing the risk of encountering major issues later in the development process. AI can also help in identifying tradeoffs between different design alternatives, such as balancing non-functional requirements, for example, balancing performance with maintainability or scalability. The ability to foresee potential problems in design helps developers make more informed, data-driven decisions, leading to more robust and efficient systems.

Explainability and Trust in AI Design Tools. A critical factor in the successful integration of AI in software design is explainability. While AI systems can propose design suggestions or evaluate tradeoffs, these recommendations must be accompanied by clear, interpretable explanations. **Explainable AI (XAI)** [24] techniques are essential to build trust among developers, who must understand the reasoning behind AI-generated suggestions to make informed decisions. Research into XAI is particularly important in the context of software design, where trust and transparency are paramount. AI should be able to justify its choices, explain the benefits and drawbacks of each design alternative, and provide insights into the long-term implications of design decisions. This transparency will help developers feel more confident in relying on AI for complex design tasks and foster greater collaboration between humans and machines. Explainable AI is an important and active research topic in the AI community [24, 105], and more work is needed to leverage explainability techniques in the context of software design.

[10] C4 Model: https://c4model.com/

> *Software Design*: An important research challenge will be to understand how software engineers can effectively integrate AI into their design workflows, communicate with them, and interpret their suggestions. In particular, AI must provide explanations for their design suggestions to increase trust and facilitate human understanding.

3.3 Software Development and Testing

We envision that software development and testing will be intertwined, as automated testing should be conducted to verify the correctness of the components generated by AI, as well as their seamless integration into the code base. Given a set of unimplemented requirements, AI will automatically generate and test the production code, after which humans and AI will collaborate to improve and verify it. Indeed, AI is already being used to automatically generate code and its associated tests, for example, GitHub's Copilot. However, we envision several ways AI code generation will evolve.

Ensuring High-Quality Code. First, it is important that AI-generated code is correct (behaves as expected). Code generated by existing LLMs is not always correct [69]. We envision significant improvements in the correctness of AI-generated code, particularly as systems learn from the selections and refinements of generated code made by software engineers. Thus, it is also important that code generated is understandable, while humans remain in the loop. There is a risk, as more AI generated code is integrated into software systems and is fed back into training data for future code generation, that understandability will be reduced.

Other quality attributes like reliability, security, scalability, and performance will also be important as more code is generated by AI. This can be particularly challenging given that there will be large variations in the training data when it comes to these attributes. For example, research has found that AI-generated code is often insecure but can be improved by effective prompt engineering [31]. Thus, more work is needed to ensure that these quality attributes are considered in AI code generation models and good prompt engineering will continue to be important. Education in software engineering will need to integrate prompt engineering into the curriculum.

AI-Assisted Updates. It is well known that software requirements continuously change. Changes can occur for many reasons, including changing user needs, changing environments, or new regulations. The complexity of evolving software to keep up with these changes was recognized decades ago by Lehman [51]. Agile processes and methods have been created to enable software teams to better respond to these continuous changes [98]. We envision that, as requirements change, AI can also automatically validate the new requirements (as described in the previous section) and generate changes to the associated code and tests. A challenge here is that requirements might be too high level, and it is difficult to decompose high-level requirements into low-level implementation details. However, we believe that as AI advances, we will move further in this direction.

By producing corresponding tests, the requirements can be automatically verified to ensure that the system fulfills the changed or added requirements. This brings promise of more advanced end user software engineering [14], where users of software systems can work together with AI systems to specify changes. This could enable more customized and personalized software systems. This is enabled through the advancements we described in previous sections, particularly the ability for AI to validate requirements and produce prototype sketches. With these, stakeholders and users without technical expertise can continuously refine their requests until the desired prototypes are obtained. Automated code and test generation could then proceed to enable these personalizations. However, this also brings maintenance challenges, which we discuss further in the next section.

Sharing of Validated AI-Generated Code. An important opportunity arises from the potential sharing of low-level implementations generated by AI within the open source community. Low-level implementations could be generated as stateless and immutable APIs. The advantage is that these APIs undergo human and automated verification and testing, including security checks to mitigate vulnerabilities. This enables reuse in other projects rather than regenerating from scratch. By accessing existing databases of AI-generated APIs, AI systems can explore alternatives before generating new code. This concept parallels the notion of "APIzation" recently explored for Stack Overflow code snippets [91, 92]. However, caution must be taken to ensure adherence to open source licensing and governance when reusing or sharing AI-generated APIs.

This can also help with sustainability challenges given the high cost of code generation in regard to energy consumption [33]. We see sustainability of AI systems a key research challenge in the coming years. As AI code generation continues to mature and AI capabilities increase, code should be continually improved to ensure that quality is maintained. Research efforts can look into how this can be done while reducing energy consumption.

> *Software Development*: Ensuring generated code is correct, understandable, reliable, secure, and scalable, while also considering energy consumption of the AI models, is a key research challenge. Prompt engineering will continue to be important.

Testing will play a crucial role, as we need to ensure the correctness of the LLM-generated code and its integration with the codebase. Test cases can, of course, be created by developers, but they can also be generated automatically. The latter type of test cases will be crucial for verifying AI-generated code.

Researchers are exploring the use of LLMs to generate test cases [9, 21, 53, 58, 70, 80, 102, 108], showing promising results. Therefore, we assume that an LLM-based agent dedicated to test case generation will be included in our framework.

However, we envision that such LLM-based agents will work in combination with automated test generators (e.g., Randoop [71], EvoSuite [28], and Pynguin [60]) to improve the quality and fault detection effectiveness of the generated tests. We are already witnessing the first attempt of this combination, yielding promising results [52]. While LLMs can be somewhat effective in generating test cases [79, 112], current LLMs do not guarantee compilable or runnable test cases [112].

Therefore, an integration with traditional test generators that compile and run test cases is necessary. Additionally, the feedback from compiling and running test cases is known to be extremely useful in improving LLM-generated tests [79, 112], or automatically generate test cases in general (e.g., feedback-directed approach [71]). More research is needed to better exploit the synergy and complementarity of LLMs and traditional test case generators [52].

Indeed, generating *effective oracles* that correctly distinguish between correct and incorrect executions is crucial. We cannot expect humans to write oracles for (many) AI-generated test cases; we need automatically generated oracles. Unit test generators (e.g., Randoop [71] and Evosuite [28]) generate (regression) oracles based on the implemented behavior, not the intended one. They capture the implemented behavior of the program with assertions that predicate on the values returned by method calls and fail if a future version leads to behavioral differences. Thus, they are only useful in a regression testing scenario, and their effectiveness is usually evaluated in such a scenario [41, 82]. Regarding AI-generated code, the regression scenario is not useful as we want to expose faults in the current version of AI-generated code.

Although recent research shows that LLM-based agents for test generation often produce oracles that capture the actual program behavior instead of the expected behavior [11, 48, 115], their overall effectiveness remains limited and still far from completely addressing the oracle problem [48].

Metamorphic Testing (MT) [17] could be the key to address this challenge. MT alleviates the oracle problem by using relations among the expected outputs of related inputs as oracles [18]. Research shows that such relations, called **Metamorphic Relations (MRs)**, exist in virtually any software system [81]. MT proves highly beneficial when integrated into automated test generation, as a single MR can be applied to all test automatically generated inputs that satisfy the input relation. However, MT's automation and effectiveness depend on the availability of MRs. The automated generation or discovery of MRs presents a challenging and largely understudied problem [2, 18, 19, 81]. Only recently has the research community begun addressing MR generation from different angles [5, 6, 12, 103, 104, 113, 114]. More research is needed on MR generation [5, 6, 104] and oracle/generation improvement [39, 40, 67, 89, 90] to facilitate effective testing of AI-generated code.

Interestingly, recent research has explored using LLMs for MT of software systems [3, 84, 96], including automatically generating MRs [62, 83, 86, 94, 103]. These studies show the potential of LLMs to fully automate MT.

> *Software Testing*: The key research challenge will be to automatically generate test cases with effective oracles to verify AI-generated code.

3.4 Software Maintenance

We envision an AI-powered maintenance phase that remains constantly active in the background. The AI will monitor a wide range of external information sources about the software product and its ecosystem to proactively gather potential issues or opportunities for improvement.

Indeed, issues or maintenance opportunities are often buried in a *large amount of sources*, such as bug reports, automated alerts, error logs, discussions on developer forums, and feedback from app stores [93, 95]. While current AI tools like Snyk's DeepCode focus on real-time code analysis for security vulnerabilities and code quality, we envision future AI systems extending this capability by analyzing and aggregating insights from diverse, unstructured data sources. Such systems will be capable of autonomously extracting relevant insights, identifying potential issues, and proposing appropriate fixes or improvements to the software artifacts.

In particular, there are important ethical considerations when new product improvements and feature requests can be gathered from the crowd. The AI system should not solely focus on the most popular feature requests and issues but also those that are less popular but might target minority and disability groups [26, 63]. Further, the AI cannot simply add every feature users suggest, some consideration with the product strategy must be considered [47]. AI systems can be trained on the product strategy documents to ensure that new features align with the overall vision for the product. The strategy will need to be continuously kept up-to-date and while AI can likely assist with this, humans must be kept in the loop.

Another way AI can assist in software maintenance is by keeping track of the context and history of a software project. AI can learn from the project's history to avoid repeating mistakes from the past. Detailed comments on design decisions and bug fixes can enable this, and explainable AI, as discussed above, can automatically create such details in the future.

Additionally, software exists within an *ecosystem of external libraries*. The libraries that a project depends on may release new versions to address vulnerabilities or bugs, making it essential to keep dependencies up-to-date. However, major updates can introduce breaking changes or

compatibility issues with the existing codebase. The AI system should automate minor and patch updates but carefully assess major upgrades, which often require more adaptation due to potential incompatibilities. By analyzing the project's dependency history and identifying necessary code modifications, the AI can ensure that security and performance improvements are applied without disrupting existing functionality. Furthermore, the AI should detect and resolve static [42, 44] or behavioral [43] breaking changes, reducing manual intervention and supporting developers in maintaining stable software. Future research could build on work in automated program repair [50, 57] to further enhance these capabilities.

As described in the previous section, advancements in AI code generation will enable more sophisticated end user programming. Users can specify new features to customize and personalize their software systems, leading to challenges in terms of software maintenance due to the potential proliferation of different software versions. AI should be capable of continuously updating these personalized versions, ensuring that new features and optimizations from the main version remain accessible to users who have made customizations. Additionally, gate-keeping mechanisms will likely need to be developed to prevent personalizations from introducing security vulnerabilities or other issues.

Another potential challenge is to ensure adherence to software licenses. Currently, AI-generated code does not automatically account for the licenses of the software used in its training data, leading to concerns about potential violations. For example, *Copyleft licenses* require that any modified or extended version of the code remains under the same license and attributes the original source [13]. Tools like GitHub Copilot have faced criticism for reusing open source code without appropriate attribution or compliance with these licensing terms [8]. Companies like OpenAI, GitHub, and Microsoft argue that using publicly available code to train AI systems falls under fair use. However, this legal position remains contentious, and future regulations may require AI-generated code to explicitly include attribution and license details. Addressing these legal challenges would necessitate advancements in AI code generation systems, ensuring that they can identify and apply the correct licenses to any code they produce.

> *Software Maintenance*: The primary research challenge will be to enable AI to autonomously process and utilize a vast amount of external information effectively to identify potential issues or opportunities for improvement. The AI should achieve this while ensuring fairness in its decision-making process and adherence to strategic direction.

4 Limitations and Risks

The synergistic collaboration between software developers and AI (in particular, LLMs) offers immense benefits. For example, automating repetitive tasks or augmenting human effort can allow developers to focus on higher level design and problem-solving. However, it also comes with certain limitations and risks.

4.1 LLM-Specific Limitations and Risks

Correctness Issues. LLMs, especially in complex software engineering tasks, may "hallucinate," produce incorrect, incomplete, or misleading outputs [107]. While LLMs are continuously improving, it is expected that they will eventually hit a ceiling. As discussed in our framework, automated checking and validation of LLM-generated outputs (possibly involving non-AI tools) will be essential to address these limitations. For instance, combining LLM code generation with a compiler to verify correctness is a promising trend that is already emerging [10].

Homogeneity of Code. A potential issue in the future is that as more developers integrate AI into their workflows, LLMs may increasingly train on LLM-generated code. This could reduce the diversity of code practices over time, as LLMs tend to reaffirm patterns seen in their training data (e.g., public GitHub repositories). While this could make code more uniform and easier to understand for humans, it might also prevent exploration of alternative or potentially better coding practices. Additionally, such bias could lead to slow or insecure code being perpetuated by LLMs.

Ethical and Fairness Concerns. Being trained on large amount of data, LLMs are inherently biased toward common software practices and ways of thinking [37], which may not fully consider cultural or gender diversity. For example, previous research has found gender differences in various cognitive facets including information processing style with women more likely to prefer a comprehensive approach and have all information available before starting a task [15]. It is unknown if such cognitive diversity leads to differences in code style. This risk may be amplified by the above risk of homogeneous code, where the dominant way of thinking can be proliferated as code becomes more homogeneous. This is a general risk of AI, and addressing it is a critical challenge that requires effort from the AI community. Fine-tuning software engineering-specific LLM models to include sensitivity to ethical and fairness issues is an important step in mitigating this risk. More research is also needed to understand diversity of thought in relation to software design and code and how this can be maintained, or even broadened through increased participation, with LLMs.

Data Privacy. LLMs services like OpenAI have the potential to memorize sensitive or proprietary information and expose it to other users. Additionally, input data could be used as training data for future versions of the model [106, 109]. This is a serious risk, prompting many companies to use local or private LLM instances to ensure data privacy. We believe that local LLM instances will become increasingly common, although they raise sustainability concerns. A centralized system (e.g., OpenAI models) is typically more cost-efficient for large-scale usage. However, private LLMs can be trained or fine-tuned with project- and company-specific data, enhancing performance for specialized tasks.

4.2 AI—General Limitations and Risks

Over-reliance on AI. Relying too heavily on AI may reduce developers' problem-solving and critical thinking skills, as they become overly dependent on AI-generated solutions. This can lead to a loss of software engineering skills if humans become reliant on AI. This is of specific concern given the non-deterministic nature of AI. For example, reliance on a calculator, which was also controversial when first introduced, will always produce the same output given the same inputs, but the same is not true for AI systems. Software engineering skills are still needed to ensure the quality of software systems. Similarly, people without software engineering expertise may be overly confident in their abilities when leveraging LLMs. There is a broader risk that software developers might write less code over time, as their job could mainly become reviewing code generated by AI. This could lead to frustration and less sense of ownership over the codebase.

Malicious Activities. Bad actors might exploit LLMs to generate malicious code or automate unethical practices, posing significant security risks. This is another reason why humans must remain in control and maintain their software engineering skills.

Computational Requirements. The high computational demands of AI (especially LLMs) can increase costs and environmental impact [78]. However, recent advancements have shown that smaller, more efficient models can achieve good performance (e.g., see GPT-4o-mini[11]). We believe that the key lies in determining task complexity automatically and delegating only the most complex tasks to expensive LLM models [46].

[11] https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/

5 Conclusions

This article presented a *vision of a symbiotic partnership between AI and software developers* motivated and inspired by recent advances in AI. This article also discussed some key research challenges that need to be addressed by the software engineering community. While this article focuses on specific software engineering challenges, it is essential to acknowledge broader AI-related concerns, such as security, safety, bias, and privacy. Although not covered here, these issues are crucial but fall more within the domain of the AI community, and hopefully will be addressed soon.

We cannot ignore the opportunities that lie ahead. Nor should we disregard the concerns associated with them. Specifically, we must exercise caution *against over-reliance on AI*. While the next generations of software engineers should be trained in prompt engineering and AI, this should not overshadow the necessity of core software engineering knowledge. Human judgment remains indispensable for critically assessing AI-generated artifacts. It is crucial to emphasize again that AI serves as a tool to enhance developers' productivity and cannot (in the near future) replace humans. Putting too much trust on the software artifacts generated by AI can have serious repercussions on the quality and safety of our software systems [72, 110].

This article serves also as a *call to arms for our community*. We need multi-disciplinary collaborations across our community to address the key challenges and achieve the envisioned symbiotic partnership between human developers and AI. While our vision is ambitious, we believe that a 5- to 10-year timeframe is reasonable for realizing it.

References

[1] Esra A. Abdelnabi, Abdelsalam M. Maatuk, Tawfig M. Abdelaziz, and Salwa M. Elakeili. 2020. Generating UML class diagram using NLP techniques and heuristic rules. In *Proceedings of the 2020 20th International Conference on Sciences and Techniques of Automatic Control and Computer Engineering (STA)*, 277–282. DOI: https://doi.org/10.1109/STA50679.2020.9329301

[2] John Ahlgren, Maria Eugenia Berezin, Kinga Bojarczuk, Elena Dulskyte, Inna Dvortsova, Johann George, Natalija Gucevska, Mark Harman, Maria Lomeli, Erik Meijer, et al. 2021. Testing web enabled simulation at scale using metamorphic testing. In *Proceedings of the 43rd International Conference on Software Engineering*, 140–149. DOI: https://doi.org/10.1109/ICSE-SEIP52600.2021.00023

[3] Nadia Alshahwan, Jubin Chheda, Anastasia Finogenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. 2024. Automated unit test improvement using large language models at meta. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE '24)*. ACM, 185–196. DOI: https://doi.org/10.1145/3663529.3663839

[4] Chetan Arora, John Grundy, and Mohamed Abdelrazek. 2024. Advancing requirements engineering through generative AI: Assessing the role of LLMs. In *Generative AI for Effective Software Development*. A. Nguyen-Duc, P. Abrahamsson, and F. Khomh (Eds.), Springer, 129–148.

[5] Jon Ayerdi, Valerio Terragni, Aitor Arrieta, Paolo Tonella, Goiuria Sagardui, and Maite Arratibel. 2021. Generating metamorphic relations for cyber-physical systems with genetic programming: An industrial case study. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21)*. ACM, 1264–1274. DOI: https://doi.org/10.1145/3468264.3473920

[6] Jon Ayerdi, Valerio Terragni, Gunel Jahangirova, Aitor Arrieta, and Paolo Tonella. 2024. GenMorph: Automatically generating metamorphic relations via genetic programming. *IEEE Transactions on Software Engineering* 50, 7 (2024), 1888–1900. DOI: https://doi.org/10.1109/TSE.2024.3407840

[7] G. Octo Barnett and Robert A. Greenes. 1970. High-level programming languages. *Computers and Biomedical Research* 3, 5 (1970), 488–494.

[8] Vivek Basanagoudar and Abhijay Srekanth. 2023. Copyright conundrums in generative AI: Github Copilot's not-so-fair use of open-source licensed code. *Journal of Intellectual Property Studies* 7 (2023), 58–68.

[9] Shreya Bhatia, Tarushi Gandhi, Dhruv Kumar, and Pankaj Jalote. 2024. Unit test generation using generative AI: A comparative performance analysis of autogeneration tools. In *Proceedings of the 1st International Workshop on Large Language Models for Code*, 54–61. DOI: https://doi.org/10.1145/3643795.3648396

[10] Zhangqian Bi, Yao Wan, Zheng Wang, Hongyu Zhang, Batu Guan, Fangxin Lu, Zili Zhang, Yulei Sui, Hai Jin, and Xuanhua Shi. 2024. Iterative refinement of project-level code context for precise code generation with compiler

feedback. In *Proceedings of the Findings of the Association for Computational Linguistics (ACL '24)*. ACL, 2336–2353. DOI: https://doi.org/10.18653/V1/2024.FINDINGS-ACL.138

[11] Soneya Binta Hossain and Matthew Dwyer. 2024. TOGLL: Correct and strong test oracle generation with LLMs. arXiv:2405.03786. Retrieved from https://arxiv.org/abs/2405.03786

[12] Arianna Blasi, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Antonio Carzaniga. 2021. MeMo: Automatically identifying metamorphic relations in Javadoc comments for test automation. *Journal of Systems and Software* 181 (2021), 111041. DOI: https://doi.org/10.1016/J.JSS.2021.111041

[13] Sharee L. Broussard. 2007. The copyleft movement: Creative commons licensing. *Communication Research Trends* 26, 3 (2007), 3–18.

[14] Margaret M. Burnett, Curtis R. Cook, and Gregg Rothermel. 2004. End-user software engineering. *Communications of the ACM* 47, 9 (2004), 53–58. DOI: https://doi.org/10.1145/1015864.1015889

[15] Margaret M. Burnett, Simone Stumpf, Jamie Macbeth, Stephann Makri, Laura Beckwith, Irwin Kwan, Anicia Peters, and Will Jernigan. 2016. GenderMag: A method for evaluating software's gender inclusiveness. *Interacting with Computers* 28, 6 (2016), 760–787. DOI: https://doi.org/10.1093/IWC/IWV046

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

[17] T. Y. Chen, S. C. Cheung, and S. M. Yiu. 1998. *Metamorphic Testing: A New Approach for Generating Next Test Cases*. Technical Report. Technical Report HKUST-CS98-01, Department of Computer Science, The Hong Kong University of Science and Technology.

[18] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. 2018. Metamorphic testing: A review of challenges and opportunities. *ACM Computing Surveys* 51, 1, Article 4 (Jan. 2018), 1–27. DOI: https://doi.org/10.1145/3143561

[19] Tsong Yueh Chen and T. H. Tse. 2021. New visions on metamorphic testing after a quarter of a century of inception. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 1487–1490.

[20] Leuson Da Silva, Jordan Samhi, and Foutse Khomh. 2024. ChatGPT vs LLaMA: Impact, reliability, and challenges in stack overflow discussions. arXiv:2402.08801. Retrieved from https://arxiv.org/abs/2402.08801

[21] Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and Michel C. Desmarais. 2024. Effective test generation using pre-trained large language models and mutation testing. *Information and Software Technology* 171 (2024), 107468. DOI: https://doi.org/10.1016/J.INFSOF.2024.107468

[22] Fabiano Dalpiaz, Davide Dell'Anna, Fatma Basak Aydemir, and Sercan Çevikol. 2019. Requirements classification with interpretable machine learning and dependency parsing. In *Proceedings of the 2019 IEEE 27th International Requirements Engineering Conference (RE)*, 142–152. DOI: https://doi.org/10.1109/RE.2019.00025

[23] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2024. Evaluating large language models in class-level code generation. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE, 865–865. DOI: https://doi.org/10.1145/3597503.3639219

[24] Rudresh Dwivedi, Devam Dave, Het Naik, Smiti Singhal, Rana Omer, Pankesh Patel, Bin Qian, Zhenyu Wen, Tejal Shah, Graham Morgan, et al. 2023. Explainable AI (XAI): Core ideas, techniques, and solutions. *ACM Computing Surveys* 55, 9 (2023), 1–33. DOI: https://doi.org/10.1145/3561048

[25] Christof Ebert and Panos Louridas. 2023. Generative AI for software practitioners. *IEEE Software* 40, 4 (2023), 30–38. DOI: https://doi.org/10.1109/MS.2023.3265877

[26] Marcelo Medeiros Eler, Leandro Orlandin, and Alberto Dumont Alves Oliveira. 2019. Do Android app users care about accessibility? An analysis of user reviews on the Google play store. In *Proceedings of the 18th Brazilian Symposium on Human Factors in Computing Systems*, 1–11. DOI: https://doi.org/10.1145/3357155.3358477

[27] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. 2023. Large language models for software engineering: Survey and open problems. In *Proceedings of the 2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*. IEEE, 31–53. DOI: https://doi.org/10.1109/ICSE-FOSE59343.2023.00008

[28] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, 416–419.

[29] Maurizio Gabbrielli, Simone Martini, and Saverio Giallorenzo. 2023. *Programming Languages: Principles and Paradigms, Second Edition*. Springer. DOI: https://doi.org/10.1007/978-3-031-34144-1

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

In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE '24)*. ACM, 1–13. DOI: https://doi.org/10.1145/3597503.3608134

[31] Stefan Goetz and Andreas Schaad. 2024. "You still have to study"—On the security of LLM generated code. arXiv:2408.07106. Retrieved from https://arxiv.org/abs/2408.07106

[32] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh. 2017. Program synthesis. *Foundations and Trends® in Programming Languages* 4, 1–2 (2017), 1–119.

[33] Lianghong Guo, Yanlin Wang, Ensheng Shi, Wanjun Zhong, Hongyu Zhang, Jiachi Chen, Ruikai Zhang, Yuchi Ma, and Zibin Zheng. 2024. When to stop? Towards efficient code generation in LLMs with excess token prevention. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24)*, 1073–1085. DOI: https://doi.org/10.1145/3650212.3680343

[34] Junda He, Christoph Treude, and David Lo. 2024. LLM-based multi-agent systems for software engineering: Vision and the road ahead. arXiv:2404.04834. Retrieved from https://arxiv.org/abs/2404.04834

[35] Wenpin Hou and Zhicheng Ji. 2024. A systematic evaluation of large language models for generating programming code. arXiv:2403.00894. Retrieved from https://arxiv.org/abs/2403.00894

[36] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2023. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology* 33, 8 (2023), 1–79.

[37] Dong Huang, Qingwen Bu, Jie Zhang, Xiaofei Xie, Junjie Chen, and Heming Cui. 2023. Bias assessment and mitigation in LLM-based code generation. arXiv:2309.14345. Retrieved from https://arxiv.org/abs/2309.14345

[38] Yuan Huang, Yinan Chen, Xiangping Chen, Junqi Chen, Rui Peng, Zhicao Tang, Jinbo Huang, Furen Xu, and Zibin Zheng. 2024. Generative software engineering. arXiv:2403.02583. Retrieved from https://arxiv.org/abs/2403.02583

[39] Gunel Jahangirova, David Clark, Mark Harman, and Paolo Tonella. 2016. Test oracle assessment and improvement. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA '16)*. ACM, 247–258. DOI: https://doi.org/10.1145/2931037.2931062

[40] Gunel Jahangirova, David Clark, Mark Harman, and Paolo Tonella. 2019. An empirical validation of oracle improvement. *IEEE Transactions on Software Engineering* 47, 8 (2019), 1708–1728. DOI: https://doi.org/10.1109/TSE.2019.2934409

[41] Gunel Jahangirova and Valerio Terragni. 2023. SBFT tool competition 2023—Java test case generation track. In *Proceedings of the 2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)*, 61–64. DOI: https://doi.org/10.1109/sbft59156.2023.00025

[42] Dhanushka Jayasuriya, Samuel Ou, Saakshi Hegde, Valerio Terragni, Jens Dietrich, and Kelly Blincoe. 2024. An extended study of syntactic breaking changes in the wild. *Empirical Software Engineering* 30, 2, (2024), 1–44.

[43] Dhanushka Jayasuriya, Valerio Terragni, Jens Dietrich, and Kelly Blincoe. 2024. Understanding the impact of APIs behavioral breaking changes on client applications. In *Proceedings of the 1st ACM on Software Engineering (PACMSE)*, 1238–1261.

[44] Dhanushka Jayasuriya, Valerio Terragni, Jens Dietrich, Samuel Ou, and Kelly Blincoe. 2023. Understanding breaking changes in the wild. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 1433–1444.

[45] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. Inferfix: End-to-end program repair with LLMs. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 1646–1656.

[46] Gurusha Juneja, Subhabrata Dutta, Soumen Chakrabarti, Sunny Manchanda, and Tanmoy Chakraborty. 2023. Small language models fine-tuned to coordinate larger language models improve complex reasoning. arXiv:2310.18338. Retrieved from https://arxiv.org/abs/2310.18338

[47] Eric Knauss, Daniela Damian, Alessia Knauss, and Arber Borici. 2014. Openness and requirements: Opportunities and tradeoffs in software ecosystems. In *Proceedings of the 2014 IEEE 22nd International Requirements Engineering Conference (RE)*. IEEE, 213–222.

[48] Michael Konstantinou, Renzo Degiovanni, and Mike Papadakis. 2024. Do LLMs generate test oracles that capture the actual or the expected program behaviour? arXiv:2410.21136. Retrieved from https://arxiv.org/abs/2410.21136

[49] Maxime Lamothe, Yann-Gaël Guéhéneuc, and Weiyi Shang. 2021. A systematic review of API evolution literature. *ACM Computing Surveys* 54, 8 (2021), 1–36.

[50] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Communications of the ACM* 62, 12 (2019), 56–65. DOI: https://doi.org/10.1145/3318162

[51] Manny M. Lehman. 1996. Laws of software evolution revisited. In *European Workshop On Software Process Technology*. C. Montangero (Ed.), Springer, 108–124.

[52] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. 2023. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 919–931.

[53] Yihao Li, Pan Liu, Haiyang Wang, Jie Chu, and W. Eric Wong. 2024. Evaluating large language models for software testing. *Computer Standards and Interfaces* 93 (2024), 103942.

[54] Jenny T. Liang, Chenyang Yang, and Brad A. Myers. 2024. A large-scale survey on the usability of AI programming assistants: Successes and challenges. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. ACM, New York, NY, 1–13. DOI: https://doi.org/10.1145/3597503.3608128

[55] Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. 2024. Large language model-based agents for software engineering: A survey. arXiv:2409.02977. Retrieved from https://arxiv.org/abs/2409.02977

[56] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024. Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems* 36 (2024), 21558–21572.

[57] Kui Liu, Li Li, Anil Koyuncu, Dongsun Kim, Zhe Liu, Jacques Klein, and Tegawendé F. Bissyandé. 2021. A critical review on the evaluation of automated program repair systems. *Journal of Systems and Software* 171 (2021), 110817.

[58] Andrea Lops, Fedelucio Narducci, Azzurra Ragone, Michelantonio Trizio, and Claudio Bartolini. 2024. A system for automated unit test generation using large language models and assessment of generated test suites. arXiv:2408.07846. Retrieved from https://arxiv.org/abs/2408.07846

[59] Sebastian Lubos, Alexander Felfernig, Thi Ngoc Trang Tran, Damian Garber, Merfat El Mansi, Seda Polat Erdeniz, and Viet-Man Le. 2024. Leveraging LLMs for the quality assurance of software requirements. In *Proceedings of the 2024 IEEE 32nd International Requirements Engineering Conference (RE)*. IEEE, 389–397.

[60] Stephan Lukasczyk and Gordon Fraser. 2022. Pynguin: Automated unit test generation for python. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering*, 168–172.

[61] Qinyu Luo, Yining Ye, Shihao Liang, Zhong Zhang, Yujia Qin, Yaxi Lu, Yesai Wu, Xin Cong, Yankai Lin, Yingli Zhang, et al. 2024. RepoAgent: An LLM-powered open-source framework for repository-level code documentation generation. arXiv:2402.16667. Retrieved from https://arxiv.org/abs/2402.16667

[62] Quang-Hung Luu, Huai Liu, and Tsong Yueh Chen. 2023. Can ChatGPT advance software testing intelligence? An experience report on metamorphic testing. arXiv:2310.19204. Retrieved from https://arxiv.org/abs/2310.19204

[63] Saurabh Malgaonkar, Sherlock A Licorish, and Bastin Tony Roy Savarimuthu. 2022. Prioritizing user concerns in app reviews—A study of requests for new features, enhancements and bug fixes. *Information and Software Technology* 144 (2022), 106798.

[64] Garima Malik, Mucahit Cevik, Devang Parikh, and Ayse Basar. 2024. Supervised semantic similarity-based conflict detection algorithm: S3CDA. arXiv:2206.13690. Retrieved from https://arxiv.org/abs/2206.13690

[65] Ke Mao, Licia Capra, Mark Harman, and Yue Jia. 2015. *A Survey of the Use of Crowdsourcing in Software Engineering*. Research Note RN/15/01.

[66] Nuno Marques, Rodrigo Rocha Silva, and Jorge Bernardino. 2024. Using ChatGPT in software requirements engineering: A comprehensive review. *Future Internet* 16, 6 (2024), 180.

[67] Facundo Molina, Pablo Ponzio, Nazareno Aguirre, and Marcelo Frias. 2021. EvoSpex: An evolutionary algorithm for learning postconditions. In *Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 1223–1235. DOI: https://doi.org/10.1109/ICSE43902.2021.00112

[68] Mona Nashaat and James Miller. 2024. Towards efficient fine-tuning of language models with organizational data for automated software review. *IEEE Transactions on Software Engineering* 50, 9 (2024), 2240–2253.

[69] Nhan Nguyen and Sarah Nadi. 2022. An empirical evaluation of GitHub copilot's code suggestions. In *Proceedings of the 19th International Conference on Mining Software Repositories*, 1–5.

[70] Wendkuuni C. Ouedraogo, Kader Kabore, Haoye Tian, Yewei Song, Anil Koyuncu, Jacques Klein, David Lo, and Tegawende F. Bissyande. 2024. LLMs and prompting for unit test generation: A large-scale evaluation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2464–2465.

[71] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*. IEEE, 75–84.

[72] 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 2022 IEEE Symposium on Security and Privacy (SP)*, 754–768.

[73] Kavita Philip, Medha Umarji, Megha Agarwala, Susan Elliott Sim, Rosalva Gallardo-Valencia, Cristina V. Lopes, and Sukanya Ratanotayanon. 2012. Software reuse through methodical component reuse and a methodical snippet remixing. In *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work*, 1361–1370.

[74] Chanathip Pornprasit and Chakkrit Tantithamthavorn. 2024. Fine-tuning and prompt engineering for large language models-based code review automation. *Information and Software Technology* 175 (2024), 107523.

[75] Asha Rajbhoj, Akanksha Somase, Piyush Kulkarni, and Vinay Kulkarni. 2024. Accelerating software development using generative AI: ChatGPT case study. In *Proceedings of the 17th Innovations in Software Engineering Conference*, 1–11.

[76] Eric Ribeiro, Ronan Nascimento, Igor Steinmacher, Laerte Xavier, Marco Gerosa, Hugo de Paula, and Mairieli Wessel. 2022. Together or apart? Investigating a mediator bot to aggregate bot's comments on pull requests. In *Proceedings of the 2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 434–438.

[77] Caitlin Sadowski, Kathryn T Stolee, and Sebastian Elbaum. 2015. How developers search for code: A case study. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 191–201.

[78] Siddharth Samsi, Dan Zhao, Joseph McDonald, Baolin Li, Adam Michaleas, Michael Jones, William Bergeron, Jeremy Kepner, Devesh Tiwari, and Vijay Gadepally. 2023. From words to watts: Benchmarking the energy costs of large language model inference. In *Proceedings of the 2023 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–9.

[79] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. Adaptive test generation using a large language model. arXiv:2302.06527. Retrieved from https://arxiv.org/abs/2302.06527

[80] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering* 50, 1 (2023), 85–105.

[81] S. Segura, G. Fraser, A. Sanchez, and A. Ruiz-Cortés. 2016. A survey on metamorphic testing. *IEEE Transactions on Software Engineering* 42, 9 (Sept. 2016), 805–824. DOI: https://doi.org/10.1109/TSE.2016.2532875

[82] Sina Shamshiri, René Just, José Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. 2015. Do automatically generated unit tests find real faults? An empirical study of effectiveness and challenges. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE '15)*. IEEE Press, 201–211. DOI: https://doi.org/10.1109/ASE.2015.86

[83] Seung Yeob Shin, Fabrizio Pastore, Domenico Bianculli, and Alexandra Baicoianu. 2024. Towards generating executable metamorphic relations using large language models. arXiv:2401.17019. Retrieved from https://arxiv.org/abs/2401.17019

[84] Mohammed Latif Siddiq, Joanna Cecilia Da Silva Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinícius Carvalho Lopes. 2024. Using large language models to generate junit tests: An empirical study. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, 313–322.

[85] Susan Elliott Sim, Medha Umarji, Sukanya Ratanotayanon, and Cristina V Lopes. 2011. How well do search engines support code retrieval on the web? *ACM Transactions on Software Engineering and Methodology* 21, 1 (2011), 1–25.

[86] Dananjay Srinivas, Rohan Das, Saeid Tizpaz-Niari, Ashutosh Trivedi, and Maria Leonor Pacheco. 2023. On the potential and limitations of few-shot in-context learning to generate metamorphic specifications for tax preparation software. arXiv:2311.11979. Retrieved from https://arxiv.org/abs/2311.11979

[87] Kathryn T. Stolee, Sebastian Elbaum, and Daniel Dobos. 2014. Solving the search for source code. *ACM Transactions on Software Engineering and Methodology* 23, 3 (2014), 1–45.

[88] Margaret-Anne Storey, Leif Singer, Brendan Cleary, Fernando Figueira Filho, and Alexey Zagalsky. 2014. The (R)Evolution of social media in software engineering. In *Proceedings of the Future of Software Engineering*, 100–116.

[89] Valerio Terragni, Gunel Jahangirova, Paolo Tonella, and Mauro Pezzè. 2020. Evolutionary improvement of assertion oracles. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 1178–1189.

[90] Valerio Terragni, Gunel Jahangirova, Paolo Tonella, and Mauro Pezzè. 2021. GAssert: A fully automated tool to improve assertion oracles. In *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering Companion (ICSE '21)*, 85–88.

[91] Valerio Terragni, Yepang Liu, and Shing-Chi Cheung. 2016. CSNIPPEX: Automated synthesis of compilable code snippets from Q&A sites. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 118–129.

[92] Valerio Terragni and Pasquale Salza. 2021. APIzation: Generating reusable APIs from stackoverflow code snippets. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*, 542–554.

[93] James Tizard, Peter Devine, Hechen Wang, and Kelly Blincoe. 2022. A software requirements ecosystem: Linking forum, issue tracker, and FAQs for requirements management. *IEEE Transactions on Software Engineering* 49, 4 (2022), 2381–2393. DOI: https://doi.org/10.1109/TSE.2022.3219458

[94] Christos Tsigkanos, Pooja Rani, Sebastian Müller, and Timo Kehrer. 2023. Large language models: The next frontier for variable discovery within metamorphic testing? In *Proceedings of the 2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 678–682.

[95] Simon Van Oordt and Emitza Guzman. 2021. On the role of user feedback in software evolution: A practitioners' perspective. In *Proceedings of the 2021 IEEE 29th International Requirements Engineering Conference (RE)*. IEEE, 221–232.

[96] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software testing with large language models: Survey, landscape, and vision. In *IEEE Transactions on Software Engineering* 50, 4 (2024), 911–936.

[97] Louis S. Wheatcraft and Michael J. Ryan. 2018. Communicating requirements—Effectively! In *Proceedings of the 28th INCOSE International Symposium*. Wiley Online Library, 716–732.

[98] Laurie Williams and Alistair Cockburn. 2003. Agile software development: It's about feedback and change. *Computer* 36, 6 (2003), 39–43.

[99] Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, et al. 2023. The rise and potential of large language model based agents: A survey. arXiv:2309.07864. Retrieved from https://arxiv.org/abs/2309.07864

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

[101] Chunqiu Steven Xia and Lingming Zhang. 2022. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 959–971.

[102] Zhuokui Xie, Yinghao Chen, Chen Zhi, Shuiguang Deng, and Jianwei Yin. 2023. ChatUniTest: A ChatGPT-based automated unit test generation tool. arXiv:2305.04764. Retrieved from https://arxiv.org/abs/2305.04764

[103] Congying Xu, Songqiang Chen, Jiarong Wu, Shing-Chi Cheung, Valerio Terragni, Hengcheng Zhu, and Jialun Cao. 2024. MR-Adopt: Automatic deduction of input transformation function for metamorphic testing. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, 557–569. DOI: https://doi.org/10.1145/3691620.3696020

[104] Congying Xu, Valerio Terragni, Hengcheng Zhu, Jiarong Wu, and Shing-Chi Cheung. 2024. MR-Scout: Automated synthesis of metamorphic relations from existing test cases. *ACM Transactions on Software Engineering and Methodology* 33, 6 (Apr. 2024), 1–28. DOI: https://doi.org/10.1145/3656340

[105] Feiyu Xu, Hans Uszkoreit, Yangzhou Du, Wei Fan, Dongyan Zhao, and Jun Zhu. 2019. Explainable AI: A brief survey on history, research areas, approaches and challenges. In *Proceedings of the 8th CCF International Conference on Natural Language Processing and Chinese Computing (NLPCC '19), Part II*. Springer, 563–574.

[106] Biwei Yan, Kun Li, Minghui Xu, Yueyan Dong, Yue Zhang, Zhaochun Ren, and Xiuzhen Cheng. 2024. On protecting the data privacy of large language models (LLMs): A survey. arXiv:2403.05156. Retrieved from https://arxiv.org/abs/2403.05156

[107] Jingfeng Yang, Hongye Jin, Ruixiang Tang, Xiaotian Han, Qizhang Feng, Haoming Jiang, Shaochen Zhong, Bing Yin, and Xia Hu. 2024. Harnessing the power of LLMs in practice: A survey on ChatGPT and beyond. *ACM Transactions on Knowledge Discovery from Data* 18, 6 (2024), 1–32.

[108] Lin Yang, Chen Yang, Shutao Gao, Weijing Wang, Bo Wang, Qihao Zhu, Xiao Chu, Jianyi Zhou, Guangtai Liang, Qianxiang Wang, et al. 2024. On the evaluation of large language models in unit test generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 1607–1619.

[109] Yifan Yao, Jinhao Duan, Kaidi Xu, Yuanfang Cai, Zhibo Sun, and Yue Zhang. 2024. A survey on large language model (LLM) security and privacy: The good, the bad, and the ugly. *High-Confidence Computing* 4, 2 (2024), 100211.

[110] Burak Yetiştiren, Işık Özsoy, Miray Ayerdem, and Eray Tüzün. 2023. Evaluating the code quality of AI-assisted code generation tools: An empirical study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT. arXiv:2304.10778. Retrieved from https://arxiv.org/abs/2304.10778

[111] Ralph Rowland Young. 2004. *The Requirements Engineering Handbook*. Artech House.

[112] Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. 2023. No more manual tests? Evaluating and improving ChatGPT for unit test generation. arXiv:2305.04207. Retrieved from https://arxiv.org/abs/2305.04207

[113] Bo Zhang, Hongyu Zhang, Junjie Chen, Dan Hao, and Pablo Moscato. 2019. Automatic discovery and cleansing of numerical metamorphic relations. In *Proceedings of the 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 235–245. DOI: https://doi.org/10.1109/ICSME.2019.00035

[114] J. Zhang, J. Chen, D. Hao, Y. Xiong, B. Xie, L. Zhang, and H. Mei. 2014. Search-based inference of polynomial metamorphic relations. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14)*. ACM, New York, NY, 701–712. DOI: https://doi.org/10.1145/2642937.2642994

[115] Quanjun Zhang, Weifeng Sun, Chunrong Fang, Bowen Yu, Hongyan Li, Meng Yan, Jianyi Zhou, and Zhenyu Chen. 2024. Exploring automated assertion generation via large language models. *ACM Transactions on Software Engineering and Methodology*. Retrieved from https://dx.doi.org/10.1145/3699598

[116] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. 2023. A survey of large language models. arXiv:2303.18223. Retrieved from https://arxiv.org/abs/2303.18223

[117] Hao Zhong and Hong Mei. 2017. An empirical study on API usages. *IEEE Transactions on Software Engineering* 45, 4 (2017), 319–334.

[118] Albert Ziegler, Eirini Kalliamvakou, X. Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. 2024. Measuring GitHub Copilot's impact on productivity. *Communications of the ACM* 67, 3 (2024), 54–63.

Received 27 March 2024; revised 16 December 2024; accepted 18 December 2024