Contents lists available at ScienceDirect

BenchCouncil Transactions on Benchmarks, Standards and Evaluations

journal homepage: www.keaipublishing.com/en/journals/benchcouncil-transactions-onbenchmarks-standards-and-evaluations/

Full Length Article

LLMs: A game-changer for software engineers?

Md. Asraful Haque

Computational Unit, Z.H. College of Engineering & Technology, Aligarh Muslim University, Aligarh-202002, India

ARTICLE INFO

Keywords:
Software engineering
Large language model
AI tools
Coding
Testing
Debugging

ABSTRACT

Large Language Models (LLMs) like GPT-3 and GPT-4 have emerged as groundbreaking innovations with capabilities that extend far beyond traditional AI applications. These sophisticated models, trained on massive datasets, can generate human-like text, respond to complex queries, and even write and interpret code. Their potential to revolutionize software development has captivated the software engineering (SE) community, sparking debates about their transformative impact. Through a critical analysis of technical strengths, limitations, real-world case studies, and future research directions, this paper argues that LLMs are not just reshaping how software is developed but are redefining the role of developers. While challenges persist, LLMs offer unprecedented opportunities for innovation and collaboration. Early adoption of LLMs in software engineering is crucial to stay competitive in this rapidly evolving landscape. This paper serves as a guide, helping developers, organizations, and researchers understand how to harness the power of LLMs to streamline workflows and acquire the necessary skills.

1. Introduction

Software engineering (SE) processes refer to the structured set of activities involved in the development of software systems, including requirements analysis, design, coding, testing, deployment, and maintenance. These processes ensure that software is built systematically and meets user needs while maintaining quality and reliability [1]. Software development follows various models such as the Waterfall, Agile, or DevOps, each outlining different approaches to these phases. Software engineering can be costly and time-consuming for several factors related to the complexity, labor intensity, and long-term maintenance requirements. Fig. 1 illustrates a typical breakdown of the effort or cost allocation throughout the various phases of software development life cycle (SDLC) [2,3]. The primary objective of software engineering is to develop high-quality software at a minimal cost. The software industry faces numerous challenges in developing reliable software, particularly as systems become increasingly complex [4]. The demand for faster development cycles, high-quality code, and the ability to handle large-scale systems has driven the adoption of new tools and technologies. Among these, Large Language Models (LLMs) have emerged as a powerful force, automating and optimizing various aspects of the software engineering process [5]. Large language models are state-of-the-art NLP tools that have been trained on massive amounts of data, allowing them to generate human-like responses and understand complex language patterns. They have gained immense popularity in recent years because it makes a lot of things easier and quicker. They have the potential to revolutionize various industries and transform the way we interact with technology. They have demonstrated impressive capabilities that are directly applicable to software engineering [6,7]. Some of the key functions include code generation, debugging, testing etc. The integration of LLMs into software engineering (SE) is transforming traditional practices in multiple ways. From altering how developers write, review, and maintain code to revolutionizing collaboration within teams, LLMs are reshaping the landscape of SE [8–10]. The impact of LLMs on software engineering tools and platforms is evident in the growing trend of LLM-powered IDEs. These environments now offer intelligent code suggestions, natural language queries, and automated refactoring, making development more intuitive. While there's much excitement about LLMs in software engineering, significant concerns remain regarding their practical use and ethical implications [11]. LLMs lack true comprehension of the logic behind code, making them prone to generating incorrect or insecure outputs. Additionally, the adoption of these models also brings challenges related to ethics, job roles, and the need for careful human oversight. Thus the question remains: Are these capabilities sufficient to significantly transform the software engineering industry?

In this paper, we aim to explore the transformative potential of Large Language Models in software engineering, assessing whether they represent an overhyped trend or a disruptive innovation capable of reshaping the field. We will delve into the technical strengths and

E-mail address: md_asraf@zhcet.ac.in.

https://doi.org/10.1016/j.tbench.2025.100204
Received 4 March 2025; Received in revised form 28 April 2025; Accepted 19 May 2025
Available online 19 May 2025

limitations of LLMs, examine real-world case studies, and discuss the ethical considerations that come with the adoption of AI-driven development tools. Through this comprehensive analysis, we seek to provide a balanced perspective on the role of LLMs in modern software engineering practices.

2. Understanding large language models

Large language models (LLMs) are built on the transformative power of the transformer architecture, a model introduced by Vaswani et al. in 2017 [12] that has since become the foundation of many advanced LLMs. The transformer architecture, unlike its predecessors like recurrent neural networks (RNNs) and long short-term memory (LSTM) networks, excels at handling long-range dependencies in data through its self-attention mechanism. This mechanism enables the model to understand and weigh relationships between all tokens in a sequence simultaneously, rather than processing them in order. This ability to capture both local and global context makes transformers highly effective for tasks that require understanding the structure and flow of text or code. In the context of software engineering, this allows LLMs to not only generate code based on natural language prompts but also to understand the intricate relationships between different parts of a codebase, which is crucial for complex tasks like debugging, code completion, and refactoring. The self-attention mechanism is a key innovation that empowers LLMs to efficiently determine the importance of different parts of input data, whether in a sentence or in a block of code. This helps LLMs better understand the context of programming languages, allowing them to predict the next steps in coding processes or provide useful suggestions during the development cycle. Another vital aspect of transformer models is positional encoding, which helps maintain the order of input data — a necessary feature when processing sequences like code, where the position of elements is critical to functionality. The combination of self-attention and positional encoding allows LLMs to process code sequences with an understanding of both immediate context and overall structure, thus improving their performance in code generation and related tasks.

The development of LLMs involves mainly three stages: pre-training, fine-tuning and reinforcement learning with Human Feedback [13–15]. In the pre-training phase, LLMs are exposed to vast amounts of textual and coded data, learning general language patterns, coding structures, and syntax from diverse sources such as books, websites, and open-source code repositories. The scope of this pre-training enables LLMs to acquire a broad understanding of multiple programming languages and frameworks, making them versatile in handling different software engineering tasks. Once pre-training is complete, the model undergoes fine-tuning on specific datasets tailored to the target application, refining its ability to perform tasks in specialized areas such as web development, cybersecurity, or enterprise software solutions. This fine-tuning process sharpens the model's ability to generate relevant, high-quality outputs in response to domain-specific inputs. At the end, reinforcement learning is used to further enhance the model's performance by interacting with an environment and receiving human feedback. The feedback, in the form of ratings, rankings, or corrections, is used as a reward signal to guide the model's learning. This is an iterative process and continues until the model meets the desired standards, at which point it can be used in real-world applications. A typical training process of OpenAI's ChatGPT has been shown in Fig. 2 [16]. By leveraging the advantages of pre-training, fine-tuning and RLHF, LLMs become proficient in understanding not only the general syntax and structure of code but also in adapting to specialized coding practices and conventions. This allows LLMs to assist with software engineering tasks such as code generation, debugging, and even testing, making them valuable tools for developers working across a variety of programming languages and problem domains. Despite these strengths, however, LLMs still face challenges, particularly when dealing with complex logic or novel problems outside of their training data. Nevertheless, their advanced architecture and training methodologies have positioned them as powerful, versatile tools in the field of software engineering.

A significant number of LLMs are already in use. Table 1 provides a brief overview of some well-known models [16–22]. The future holds promise for even more powerful and advanced LLMs.

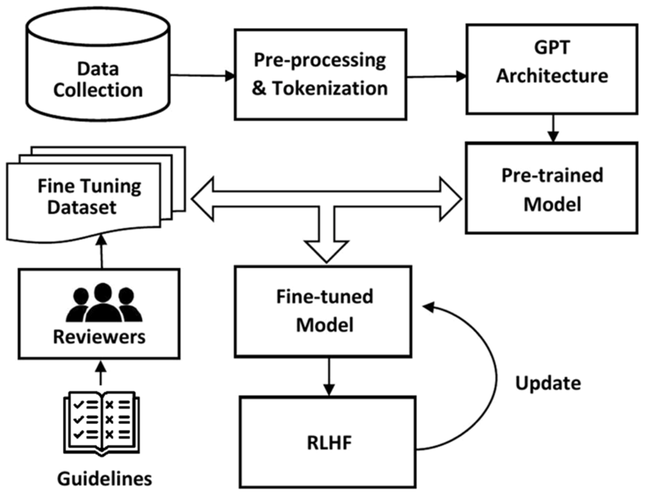
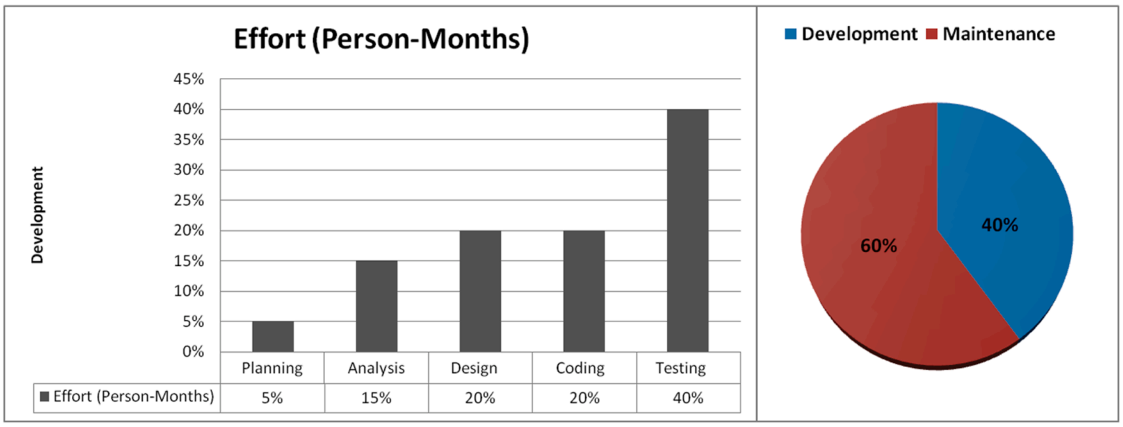


Fig. 2. ChatGPT training process.



	Planning	Analysis	Design	Coding	Testing
■ Effort (Person-Months)	5%	15%	20%	20%	40%

Fig. 1. Typical effort distribution at different phases of SDLC.

Table 1
A brief history of some prominent LLMs.

LLMs	Release Date	Developer	Model Size	
			Number of Parameters	Dimension ($L \times H$)*
BERT	October-2018	Google	110 billion (Base Model)	$L = 12$, $H = 12$ (Base Model)**
GPT-2	February-2019	OpenAI	1.5 billion	$L = 12$, $H = 12$ (Small Version)**
XLNet	June-2019	Google & CMU	110 million (Base Model)	$L = 12$, $H = 12$ (Base Model)**
T5	October-2019	Google	11 billion	$L = 12$, $H = 12$ (Small Version)**
GPT-3	June-2020	OpenAI	175 billion	$L = 96$, $H = 96$
Codex	August-2021	OpenAI	12 billion	$L = 24$, $H = 32$
PaLM	April-2022	Google	540 billion	$L = 118$, $H = 128$
GALACTICA	November-2022	Meta AI	120-billion	$L = 80$, $H = 96$
LLaMA	February-2023	Meta AI	65 billion	$L = 80$, $H = 64$
GPT-4	March-2023	OpenAI	1.76 trillion	Details undisclosed
Gemini 1.5	May-2024	Google DeepMind	Details undisclosed	Details undisclosed

* L=Number of layers, H=Number of attention heads.
** These models have different variants.

3. Technical strengths and benefits of LLMs in SE

LLMs have brought transformative potential to software engineering, providing a suite of technical strengths and benefits that can drastically enhance productivity, code quality, and innovation (Fig. 3). From improving code generation to automating complex documentation tasks, LLMs are reshaping how developers approach various phases of the software development lifecycle. While the “state-of-the-art (SOTA)” pushes toward autonomous software development, the “state-of-the-practice (SOTP)” is more about augmented intelligence — enhancing human developers rather than replacing them. The SOTA refers to the bleeding edge of what LLMs can do under optimal conditions, typically in research environments or advanced use cases, whereas the SOTP is the reality of how LLMs are being used today by software engineers in real-world environments. Below is a detailed exploration of the technical strengths and benefits of LLMs in software engineering (SE).

3.1. Code generation

One of the most prominent uses is code generation, where models like GitHub Copilot, powered by OpenAI’s Codex, allow developers to describe the functionality they need in natural language, and the LLM generates relevant code snippets [23]. This not only speeds up the coding process but also minimizes repetitive tasks, enabling developers to focus on more complex aspects of software design and architecture [24,25]. The benefit here is a marked improvement in productivity, as LLMs assist in automating routine coding activities like writing boilerplate code, implementing standard algorithms, or creating simple data structures. Moreover, LLM-driven code generation is highly versatile across different programming languages, offering cross-language flexibility that is particularly useful in polyglot development environments where multiple languages are used. The ability to generate code across Python, JavaScript, Java, C++, and other languages adds immense value, reducing the need for developers to switch contexts or master multiple languages to complete tasks efficiently [26,27].

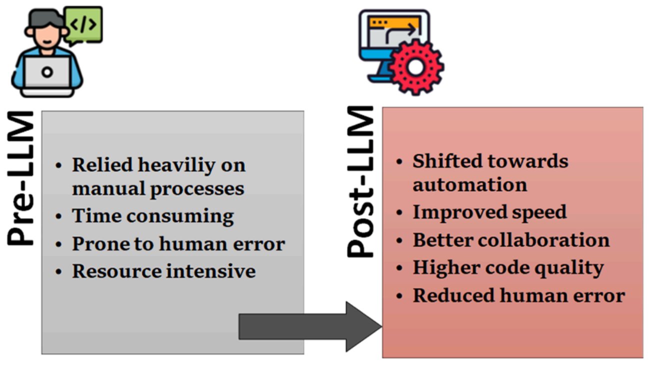


Fig. 3. Revolution in SE practices.

- State-of-the-Art: Advanced models like GPT-4, Code Llama, StarCoder, and Claude can synthesize entire functions, classes, and even small applications directly from natural language prompts. These models have achieved high benchmark scores, consistently outperforming earlier generations in evaluations like HumanEval, MBPP, and MultiPL-E, with top models surpassing 50–70 % accuracy on competitive programming tasks [23,28]. In addition, a new frontier is emerging with the development of autonomous code agents, such as Auto-GPT, Smol Developer, and Devin (from Cognition AI), which attempt to autonomously generate multi-file projects with minimal human supervision. These agents are capable of breaking down high-level goals into actionable subtasks, generating project scaffolds, and iterating toward functional software solutions with little direct developer input, pushing the limits of what automated programming can achieve.
- State-of-the-Practice: The current state of practice in industry reveals a more cautious and pragmatic adoption of LLM-based code generation tools. Platforms like GitHub Copilot, Codeium, and Amazon CodeWhisperer have seen widespread integration into professional development workflows, primarily assisting with in-line code completion, thus reducing keystrokes and improving overall developer velocity. However, LLMs are still primarily leveraged for writing boilerplate, templated code, and implementing standard patterns rather than handling complex, domain-specific logic or making architecture-level decisions. Limitations become apparent when LLMs are tasked with projects involving intricate dependencies or specialized business rules. As a result, a strong human-in-the-loop paradigm remains necessary; developers must diligently validate, test, and refine AI-generated code to ensure correctness, security, and maintainability, especially in mission-critical or production environments where even minor errors could have significant consequences.

3.2. Code review, debugging and testing

LLMs have the ability to automate code reviews and assist with bug detection [29]. These models can facilitate the knowledge required for high-quality code reviews. Even junior developers, with the assistance of LLM-powered tools, can contribute effectively to the code review process by leveraging the model’s knowledge of industry standards and best practices. Traditionally, debugging requires manual effort from developers, who inspect the code for errors, and once the error is located developers can then implement a fix to correct the error [30]. LLMs can analyze logs, error messages, and code execution paths to suggest potential causes of bugs [31,32]. This helps developers quickly pinpoint the source of an issue, reducing the time spent on manual debugging. LLMs can also suggest potential fixes based on the patterns they have learned from analyzing similar bugs in the past. This capability is particularly useful in large, complex systems where tracking down bugs can be a challenging and time-consuming task. For example, LLMs can identify common mistakes such as unhandled exceptions, resource leaks, or improper variable initializations. They can also flag potential security issues like injection vulnerabilities, insecure data handling, or incorrect encryption implementations. By catching these issues early, LLMs help developers produce more secure and robust code. Additionally, LLMs have been applied to automated testing, where they generate unit tests, identify edge cases, and suggest test cases based on the functionality described in code [33]. This application can significantly speed up the testing phase of software development, ensuring that code is rigorously tested without developers having to manually write every possible test case. LLMs' capacity to generate exhaustive test suites helps in reducing the likelihood of bugs making it to production, thus increasing the overall robustness and reliability of software systems.

- State-of-the-Art: Automated code review tools such as CodiumAI, CodeGPT, and SonarLint AI leverage LLMs to provide intelligent suggestions, identifying issues related to logic, security vulnerabilities, and adherence to coding best practices. AI-assisted debugging is also becoming highly sophisticated; for instance, GPT-4 integrations in environments like Visual Studio Code and specialized developer assistants unveiled at OpenAI DevDay can actively parse logs, diagnose errors, and recommend fixes [34]. In testing, LLMs have achieved notable success in generating not just basic unit tests but also more complex integration and property-based tests, sometimes attaining over 90 % code coverage for simple functions, as seen in models like DeepMind's AlphaCode [35,36]. These advancements signal a move towards increasingly autonomous support systems that can substantially augment developers' capabilities throughout the software lifecycle.
- State-of-the-Practice: AI tools are primarily used to augment rather than replace human-led code reviews. While they are effective at catching common mistakes and suggesting improvements, they often lack the deep contextual understanding necessary for evaluating architectural choices or domain-specific logic. False positives remain a significant concern, particularly in debugging, where LLMs sometimes propose solutions that seem plausible but fail in real-world execution. This limits the reliability of AI-suggested fixes without human verification. Moreover, while AI-generated test cases can be helpful, integration with Continuous Integration/Continuous Deployment (CI/CD) pipelines remains limited. Most teams continue to rely heavily on manual validation and traditional test frameworks, preferring to use AI-generated tests as a supplementary resource rather than fully trusting them for critical deployments.

3.3. *Language and framework agnostic*

LLMs have the remarkable ability to work across a wide variety of programming languages and frameworks, making them versatile tools in multi-language environments [37]. Because LLMs are trained on diverse datasets that include code from many different programming languages, they can switch between languages and frameworks with ease. This is especially useful for developers who work in environments that require knowledge of multiple languages, such as Python for backend services, JavaScript for frontend development, and SQL for database management. By supporting a broad range of languages and frameworks, LLMs eliminate the need for developers to switch between different coding assistants or learn new tools for each technology they use [38,39]. This contributes to a more seamless development experience and enhances overall efficiency.

- State-of-the-Art: Modern LLMs, trained on vast repositories such as GitHub and Stack Overflow, possess multilingual code understanding, allowing them to generate and translate code across a wide range of languages including Python, JavaScript, Java, $C++$, Rust, and Go. These models are not only language-flexible but also framework-aware; they can produce code that aligns with the syntax and best practices of popular frameworks such as React, Django, and Spring Boot [40]. Furthermore, cutting-edge AI-assisted refactoring tools are now capable of facilitating cross-language migration — for instance, helping transition legacy systems from COBOL to Java or modernizing codebases from Python 2 to Python 3, or even migrating JavaScript codebases to TypeScript. These advancements enable significant leaps in productivity, particularly when modernizing or maintaining large, aging code infrastructures.
- State-of-the-Practice: In real-world usage, AI-generated code often lacks deep context awareness, meaning it may not consistently adhere to team-specific coding conventions, domain-driven architectural principles, or nuanced project standards. Although LLMs can generate framework-specific code — for example, producing Django, Flask, Express, or FastAPI templates — real-world implementation typically requires manual fine-tuning and adjustment to meet production-grade requirements. Additionally, while cross-language migration tools can offer substantial initial assistance, challenges frequently arise when deep dependencies, intricate business logic, or architecture-specific constraints are involved. In such cases, human expertise remains indispensable to ensure the successful and accurate transition of complex software systems.

3.4. *Refactoring and optimization*

As software systems grow, they often accumulate technical debt, requiring refactoring and optimization to ensure long-term performance and maintainability. LLMs can assist with these processes by suggesting refactoring opportunities, such as code that can be simplified, duplicated code that can be consolidated, or outdated structures that need updating [41]. This is particularly valuable in large codebases where manually identifying areas for refactoring would be time-consuming and prone to oversight. LLMs can also help optimize code by suggesting more efficient algorithms or design patterns based on established best practices. For instance, if a developer writes a brute-force solution for a problem, the LLM might suggest a more optimal approach using dynamic programming or divide-and-conquer algorithms. Additionally, LLMs can provide performance insights, such as identifying inefficient loops, excessive memory usage, or potential bottlenecks in code execution, which helps ensure that the software remains scalable and performant as it evolves [42].

- State-of-the-Art: Automated code refactoring tools powered by LLMs — such as CodiumAI, IntelliCode, and Tabnine — can suggest enhancements aimed at improving readability, modularity, and system performance. Beyond stylistic changes, modern LLMs can engage in AI-driven performance tuning by suggesting optimizations in algorithmic complexity (Big-O improvements), recommending more efficient SQL queries, and proposing memory-optimized solutions based on recognized best practices [43,44]. Some cutting-edge models even demonstrate a level of semantic understanding of code structures, allowing them to detect "code smells" and recommend modularization strategies to improve maintainability and reduce technical debt. These advancements position LLMs as powerful partners for elevating the quality and efficiency of software development.
- State-of-the-Practice: While LLMs effectively handle simple refactoring tasks — such as renaming variables, extracting methods, and improving code readability — major architectural refactoring efforts still largely require human expertise and strategic judgment. LLMs often suffer from limited awareness of a project's full architectural context, which can result in inconsistent or suboptimal suggestions, particularly when working across large repositories with complex interdependencies. Trust remains another key issue: developers usually conduct a careful manual check of AI-proposed optimizations before adopting them, wary of potential regressions or performance declines.

3.5. Automated documentation

Generating accurate, up-to-date documentation has long been a challenge for software engineers, as it is often seen as tedious work that lags behind code changes [45]. Developers often deprioritize this due to tight deadlines or a focus on feature development. However, documentation is crucial for ensuring that code is maintainable, understandable, and transferable across teams and developers. LLMs can analyze code and automatically generate documentation for codebases, including explaining the purpose and functionality of specific functions, classes, and modules [46]. This ensures that documentation stays current and can be updated as code evolves, providing developers with easily understandable, well-structured explanations of how various parts of the system work. This capability not only improves team collaboration and knowledge sharing but also supports on-boarding processes by helping new developers quickly understand legacy codebases or complex systems. In agile environments, where requirements and implementations frequently change, the ability of LLMs to update documentation dynamically as the code evolves is an invaluable benefit.

- State-of-the-Art: State-of-the-art developments in automated documentation leverage LLMs' deep code understanding to create more contextual and helpful outputs. Modern tools like Codeium and Copilot Chat enable the generation of auto-populated docstrings, providing context-aware inline explanations that enhance code readability. Advanced LLMs can also produce natural language descriptions of APIs, offering suggestions for refining Swagger or OpenAPI specifications, thus bridging gaps between developers and API consumers. Some AI tools, like Sourcegraph Cody, push the boundaries further by summarizing codebases into readable tutorials or blog-style documentation, making complex technical information more accessible to broader audiences. These innovations are redefining how documentation is created and maintained, shifting it from a manual, error-prone task to an automated, continuous process.
- State-of-the-Practice: In current development workflows, the output of LLMs often tends to be generic, verbose, or redundant, necessitating human refinement to ensure clarity and relevance. While LLMs excel at describing low-level function behaviors, they typically struggle to capture and explain high-level architectural decisions or intricate system designs. As a result, AI-generated documentation is commonly integrated into internal knowledge bases like Confluence, Notion, or proprietary wikis, where it serves as a helpful supplement rather than a full replacement for human-authored documentation.

4. Challenges

While LLMs offer exciting possibilities in the SE domain, several technical limitations and a range of ethical challenges must be addressed:

4.1. Technical limitations

- Lack of True Understanding: LLMs do not "understand" code in the same way humans do [8,44]. While LLMs are powerful at predicting sequences based on statistical patterns learned from vast datasets, they lack a deep understanding of the underlying logic and intent behind a given piece of code. In software engineering, this is particularly problematic because coding often requires not just syntactically correct solutions, but solutions that align with specific business logic, system architecture, and performance requirements [47]. For instance, an LLM may generate syntactically correct code for a sorting algorithm but fail to account for efficiency constraints such as time complexity or memory usage, especially when these concerns are implicit in the task description. The inability of LLMs to grasp these nuances means that while they are useful for generating code snippets or suggesting fixes, developers must rigorously review and adapt their outputs to ensure they meet the functional and non-functional requirements of the system.
- Context Sensitivity: Although LLMs are good at handling short, localized contexts, they often struggle with maintaining long-term context over extended portions of a codebase [48]. Software systems are often composed of multiple files, modules, and libraries that interact with one another in complex ways. Maintaining context across such a large-scale system, where changes in one part of the codebase can have ripple effects across the system, is a challenge for LLMs. For example, an LLM may generate code that works well within a single function but fails to account for broader architectural considerations, such as how this function interacts with others in different modules or libraries. In large enterprise-scale applications, this limitation becomes even more pronounced, as developers need to track dependencies across various subsystems, which LLMs may not handle effectively. The model's understanding tends to deteriorate when it needs to work with codebases that span across multiple files or projects, resulting in incomplete or incorrect suggestions.
- Inability to Handle Novel or Rare Problems: LLMs rely heavily on patterns learned from their training data, which means they perform best when tasked with solving common or well-documented problems. However, when faced with novel or rare problems that deviate from established patterns, LLMs often struggle to produce correct or meaningful output. In software engineering, developers frequently encounter unique challenges that require creative problem-solving and a deep understanding of both the problem domain and system architecture. LLMs, constrained by the limitations of their training data, may not have seen enough similar examples to provide an adequate solution. This is especially true for cutting-edge technologies or innovative software designs that have not been widely adopted and thus are not well-represented in public datasets. Furthermore, rare edge cases, which are often the most critical and challenging parts of software development, tend to be poorly handled by LLMs due to the lack of exposure to similar situations during training.
- Computational Costs: Large Language Models (LLMs) are computationally intensive, requiring significant hardware resources for training and inference [49]. Training LLMs requires immense computational resources, including large clusters of GPUs or TPUs, leading to high financial costs. Inference, or using the trained model for tasks, can also be computationally expensive, especially for larger models. These high computational costs can be a barrier to entry for organizations, limiting their ability to adopt and utilize LLMs in their software engineering workflows. Another consideration is the energy consumption associated with running LLMs. The large-scale deployment of LLMs in software engineering environments contributes to increased energy usage, which has both economic and environmental implications [50].
- Transparency and accountability: LLMs are often seen as black-box models, meaning that their decision-making processes are not easily interpretable by users [51]. When an LLM generates code, it is not always clear how or why it arrived at a particular solution [52]. This lack of transparency becomes problematic in scenarios where LLMs make critical decisions, such as in safety-critical systems or in applications that have legal and regulatory implications. If a software failure occurs due to an LLM's suggestion, it is difficult to assign responsibility — does the fault lie with the developer, the AI, or the organization that provided the AI? This lack of clear accountability creates challenges in governance and compliance, particularly in regulated industries like finance, healthcare, and transportation. Therefore, ensuring that LLMs are explainable and that there are mechanisms in place to track and audit AI-generated outputs is essential for fostering trust and ensuring that ethical guidelines are followed.
- Security Risks: If LLMs are trained on large public datasets, including code repositories, they may inadvertently learn insecure or

vulnerable coding practices [53]. For instance, if an LLM is trained on a repository where code contains hard-coded credentials, weak encryption methods, or unpatched vulnerabilities, the model may unknowingly generate code that replicates these flaws. This becomes particularly dangerous in security-critical applications like financial software, healthcare systems, or government infrastructure. The potential for LLMs to suggest insecure code increases the burden on developers to scrutinize the model's output closely, ensuring that it adheres to industry best practices and security standards. Therefore, while LLMs can be helpful for automating routine tasks, they should not be used blindly, especially in areas where security is paramount. Continuous oversight and refinement of the training data, as well as integration with secure coding practices, are essential to mitigate these risks.

4.2. Ethical considerations

- Copyright and Intellectual Property: LLMs are trained on publicly available data, but this data may include proprietary or copyrighted code that the model can later reproduce in different contexts [54]. When LLMs generate code that closely resembles or directly replicates code from its training data, it raises serious questions about ownership and accountability. Developers using LLM-generated code may inadvertently violate copyright laws if the generated code mirrors protected material without proper attribution. This could lead to legal disputes and undermine the trust in LLMs as reliable tools in professional software development environments. To address these concerns, companies providing LLM services must implement safeguards that either filter copyrighted material during the training process or ensure that LLM-generated content is appropriately flagged for potential legal issues.
- Biases in Training Data: One of the most pressing ethical concerns surrounding LLMs is the issue of bias in training data [55]. LLMs are trained on vast datasets that include both natural language text and code from public repositories, such as GitHub, Stack Overflow, and various forums. However, these sources can contain biased, outdated, or even harmful practices. For example, if the training data includes discriminatory language or biased coding patterns (such as gender or race-based assumptions in user data processing), the LLM may learn and perpetuate these biases in its outputs. In the context of software engineering, biased code generation can lead to inequitable software solutions, unfair user experiences, or even legal and reputational risks for companies. Moreover, models trained on real-world codebases may inherit the biases of past software engineering decisions, such as assumptions about users' technical abilities or geographical location, resulting in software that does not serve all demographics equally. Addressing these biases requires careful dataset curation, as well as developing methods for identifying and mitigating bias in LLM outputs.
- Impact on the Workforce: The impact on the workforce is another ethical issue associated with the rise of LLMs in software engineering. By automating tasks like code generation, testing, and debugging, LLMs have the potential to reduce the demand for certain types of coding jobs, particularly entry-level or junior software development roles [17,55]. This could lead to job displacement for new developers or those in low-skilled positions, creating economic inequality within the industry. Additionally, reliance on LLMs may result in a deskilling of the software engineering workforce. If developers become too dependent on AI-generated code and suggestions, they may lose the ability to write complex code or troubleshoot issues independently. This could diminish the overall expertise within the field over time, affecting the quality of software and innovation. To counteract these risks, educational systems and organizations need to evolve, focusing on upskilling developers to work alongside LLMs rather than being replaced by them. Training programs should emphasize higher-order skills like software architecture, algorithm design, and critical thinking, which cannot be easily replicated by AI.

5. Case studies and recent trends

Initially, AI tools were primarily used for specific tasks like code completion and bug detection. However, the advent of LLMs has ushered in a new era of AI-powered development. This shift promises not only greater efficiency but also new possibilities in adaptive, responsive coding environments. In exploring the impact of LLMs on software engineering, it is essential to examine real-world case studies where LLMs have been applied in different software engineering (SE) environments. By analyzing diverse cases, we can understand how LLMs can be a game-changer in some situations, while potentially overhyped or insufficient in others.

5.1. GitHub Copilot (Powered by OpenAI codex)

GitHub Copilot, powered by LLM technology, has become a widely adopted tool for code generation, suggestions, and auto-completions [56–58]. It is extensively used by developers at organizations such as Microsoft, Shopify, and Datadog, along with many individual developers. A notable case of internal adoption is GitHub's own integration of Copilot into its development workflow [59,60]. This implementation led to a significant increase in developer productivity—up to 55 % in routine coding tasks. Developers experienced faster prototyping and fewer context switches, which streamlined their workflows. Copilot proved especially effective in generating boilerplate code and recurring patterns, enabling engineers to focus more on strategic and creative aspects of development.

Another compelling example comes from Shopify, where engineers utilized Copilot for developing internal tools and web applications [61–63]. The tool contributed to a faster onboarding process for junior developers and significantly reduced cognitive load during development. Moreover, the context-aware code completions offered by Copilot helped developers become more familiar with new libraries and frameworks, enhancing overall efficiency and learning outcomes within the team

5.2. Amazon CodeWhisperer

Amazon CodeWhisperer is an LLM-based tool designed to provide real-time code suggestions based on natural language prompts, seamlessly integrating with popular IDEs such as Visual Studio Code and JetBrains [64]. A prominent case study of its application involves AWS developer teams, who employed CodeWhisperer internally to automate the generation of API integration code [65]. This implementation led to a reduction in repetitive coding time by approximately 40–50 %, significantly improving developer efficiency. One of the key benefits highlighted by the teams was the tool's high accuracy in generating code specific to AWS SDKs and services, which substantially reduced the need for frequent documentation lookups, thereby streamlining the overall development workflow.

5.3. Tabnine

Tabnine is an LLM-based tool that offers predictive code completions through either local or cloud-hosted models, making it a suitable solution for privacy-focused environments and organizations requiring team-specific model tuning [66]. A case study [67] describes how Tabnine uses Google Cloud to deliver its AI-powered coding tool to one million users. Tabnine's ML models, which help developers autocomplete about 30 % of their code, rely on Google Cloud's GPUs and Google Kubernetes Engine for scalability and performance. Tabnine values its open-source commitment, which aligns with Google Cloud's dedication to the open-source community. They also value the support they have received from Google Cloud specialists.

5.4. Codium (Now qodo)

Codium uses AI to provide intelligent code completions and automate test generation. It emphasizes code quality and security, offering on-premise deployment options. It aims to boost developer productivity by streamlining coding workflows. It supports over 70 programming languages, and integrates seamlessly with various IDEs and web editors [68]. It is a lightweight alternative to GitHub Copilot. One of its key selling points is that it is available for free to both individuals and teams, making it an accessible solution for a wide range of users. A notable case study involves "Clearwater Analytics", a fintech SaaS company prioritizing data security, who adopted Qodo (formerly Codium) to enhance developer productivity [69]. Faced with the challenge of maintaining stringent security while leveraging AI-powered coding assistance, they chose Qodo for its unique ability to be deployed within their Enterprise VPC, ensuring code privacy. Developers experienced immediate productivity gains with Qodo's code completion capabilities, resulting in significant time and cycle savings. The successful implementation was supported by dedicated Qodo team support and training, and the rapid integration of new features like chat integration.

5.5. Replit Ghostwriter

Replit Ghostwriter, an LLM-powered IDE tool, is revolutionizing coding accessibility and efficiency, particularly for students and beginner developers. Seamlessly integrated into Replit's browser-based platform, it accelerates learning by providing real-time code generation, explanation, and natural language-to-code translation [70]. Educational institutions have witnessed significant improvements in student confidence and assignment completion through its interactive support, reducing reliance on instructors. Beyond education, Ghostwriter boosts productivity for experienced developers by automating tasks, contributing to Replit's substantial user growth, which surged from 10 million to over 20 million within a year [71]. Ultimately, Ghostwriter democratizes coding, serving as a powerful learning and productivity tool that's poised to expand its impact as AI technology advances.

5.6. Sourcegraph Cody

Sourcegraph Cody is an AI-powered tool designed to enhance code navigation and documentation understanding, seamlessly integrated with Sourcegraph's code intelligence platform. Leidos, a science and technology company facing the challenge of enhancing developer productivity within a complex, security-conscious environment, adopted Sourcegraph Cody [72]. They found Cody's context-aware assistance and flexible LLM integration to be key differentiators, enabling significant time savings in code understanding, documentation, and debugging. Notably, Cody drastically reduced the time spent answering teammate questions by 75 % and cut code orientation time on legacy systems by 50 %. This resulted in increased efficiency in modernizing and migrating legacy code, with tasks previously taking sprints being completed in minutes. Leidos's experience demonstrates Cody's effectiveness in improving developer workflows, particularly in large, complex codebases, and its ability to maintain high security standards.

These case studies reflect that software industries worldwide are leveraging AI tools to streamline processes, increase efficiency, and foster creativity in problem-solving. Table 2 indicates that LLMs can generate code and suggest improvements quickly, but they often lack the precision, ethical insight, and contextual understanding that human developers provide. The AI Index 2024 Annual Report [73] highlights that software developers are among the professionals most likely to incorporate AI in their work. As AI's role within the economy grows, understanding how developers use and view AI is becoming essential. Stack Overflow, the Q&A platform for programmers, runs an annual survey targeting developers. For the first time in 2023, this survey gathered insights from over 90,000 developers — featured questions on usage of AI tools [73]. It explored how developers employ these tools, which ones they prefer, and their overall perceptions of them. Table 3 shows the developers' preferences for using AI tools in software engineering tasks. Fig. 4 is the graphical representation of Table 3.

The survey was taken in May 2023, thus it may not reflect the availability of more recent AI technologies such as Gemini and Claude 3. The other findings of that survey were as follows (Fig. 5):

- Most popular AI developer tool among professional developers, 2023 is GitHub Copilot.
- Most popular AI search tool among professional developers, 2023 is ChatGPT.
- Most popular cloud platform among professional developers, 2023 is Amazon Web Services.
- Developers cited higher productivity (32.8 %), quicker learning (25.2 %), and increased efficiency (25.0 %) as the top benefits of AI tools in their work.

Table 2
Comparative analysis.

LLM-powered Tools	Key Impacts	Challenges
GitHub Copilot	- Dramatically accelerates coding. - Boosts creativity by suggesting patterns developers may not think of. - Helps junior developers produce higher-quality code.	- Sometimes generates insecure or inefficient code. - Risk of "over-relying" without understanding the logic. - Licensing/legal concerns (e.g., code originality).
Amazon CodeWhisperer	- Stronger emphasis on secure coding (e.g., encryption, authentication). - Seamless AWS service integrations save time. - Good for enterprise-grade cloud apps.	- Biased toward AWS ecosystem, less useful for non-AWS projects. - Suggestions can sometimes be more verbose than necessary. - Less flexible across diverse programming stacks.
Tabnine	- Highly efficient for boilerplate and repetitive code. - Minimal learning curve — very easy to integrate. - Helps developers "think less" about syntax.	- Limited "deep" understanding of project-specific logic. - Doesn't recommend security or performance improvements. - Can sometimes offer shallow or redundant completions.
Codium AI	- Greatly improves code quality through auto-generated tests. - Encourages a testing culture (important for scaling teams). - Helps identify hidden bugs early.	- Test quality can vary depending on code complexity. - Not a replacement for writing well-thought-out manual tests. - Might generate overly simple test cases if not fine-tuned.
Replit Ghostwriter	- Instant environment setup saves huge time (especially for quick experiments). - Ideal for prototyping new ideas without local dependencies. - Very beginner-friendly (low barrier to entry).	- Limited control for large, complex project structures. - Not suitable for full-scale production codebases. - Dependency on Replit ecosystem for best results.
Sourcegraph Cody	- Makes navigating and understanding huge codebases faster. - Helps teams maintain consistency across large projects. - Reduces onboarding time for new developers.	- Requires setting up or connecting to indexed repositories. - Effectiveness can drop if code comments/documentations are poor. - Querying the system effectively requires some learning.

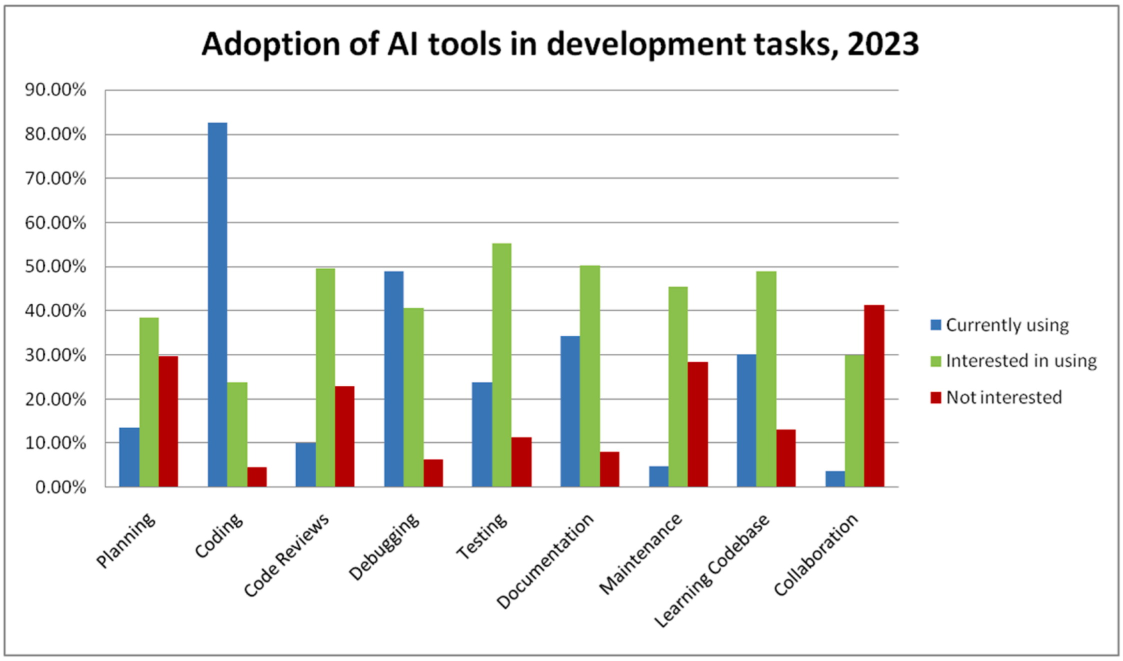


Fig. 4. Developers' preferences for using AI-tools in development tasks, 2023.

Table 3
Developers' preferences for using AI tools.

Development Tasks	Currently using	Interested in using	Not interested
Planning	13.52 %	38.54 %	29.77 %
Coding	82.55 %	23.72 %	4.48 %
Code Reviews	10.09 %	49.51 %	22.95 %
Debugging	48.89 %	40.66 %	6.37 %
Testing	23.87 %	55.17 %	11.44 %
Documentation	34.37 %	50.24 %	8.07 %
Maintenance	4.74 %	45.44 %	28.33 %
Learning Codebase	30.10 %	48.97 %	13.09 %
Collaboration	3.65 %	29.98 %	41.38 %

GitHub also conducted a survey [74] from February 26 to March 18, 2024, among 2000 non-student, corporate respondents in the United States, Brazil, India, and Germany who are not managers and work for organizations with 1000 or more employees. According to the survey, developers are increasingly integrating AI tools, with the majority of respondents reporting that AI improves their productivity and coding skills. Fig. 6 represents the respondents view on the benefits of AI tools. It highlights that popularity and use of AI tools varies by region. Fig. 7 displays the current usage of AI coding tools against the corporate endorsement for AI-driven coding. The survey respondents reported that AI tools boost productivity, freeing them up to focus on strategic tasks like system design and client collaboration. To fully leverage AI, organizations should integrate it into every phase of development. AI isn't a job replacement but an enhancer of human creativity.

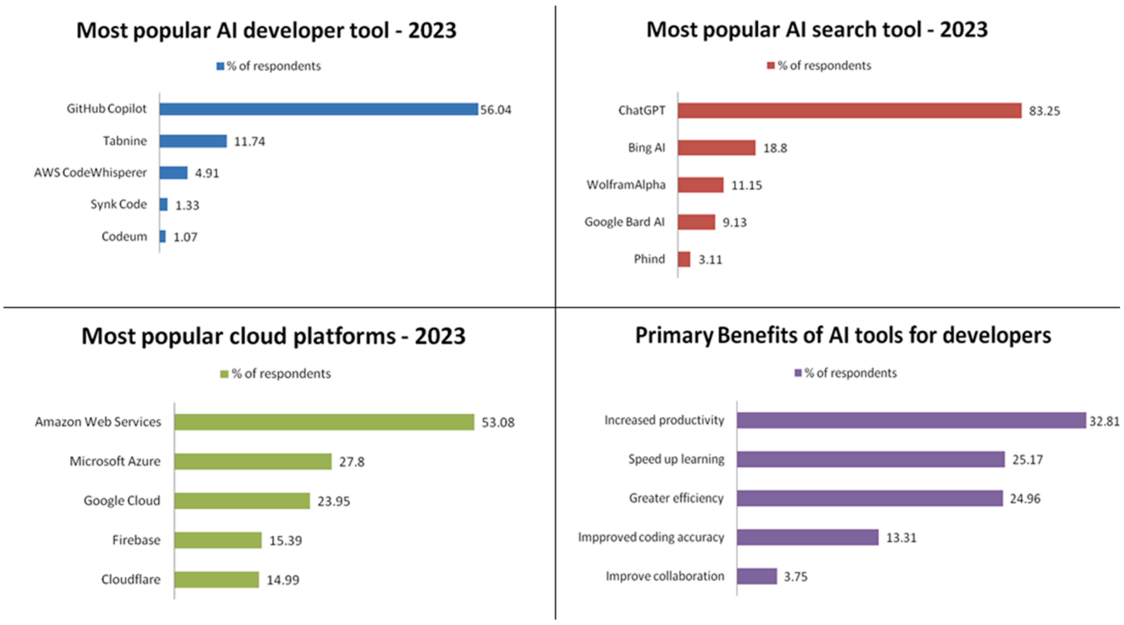


Fig. 5. Popularity of AI-tools among professional developers, 2023.

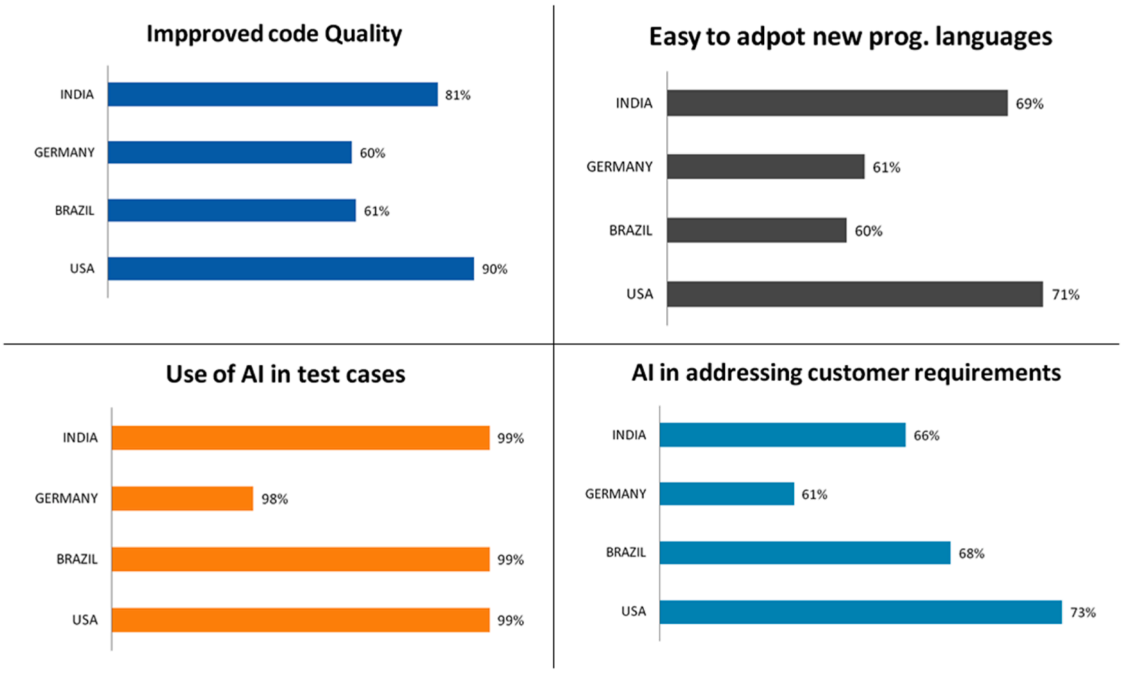


Fig. 6. Respondents view on the benefits of AI tools.

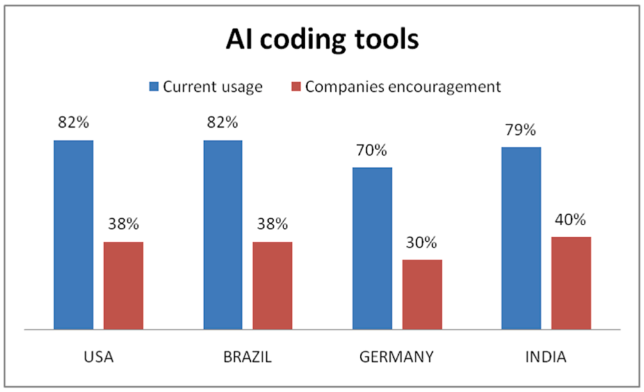


Fig. 7. Usage of AI tools vs. Companies encouragement.

6. Future directions and research opportunities

As Large Language Models (LLMs) continue to evolve and become more deeply integrated into software engineering (SE) processes, the future of this technology holds immense potential. However, there are several areas that still require further exploration, development, and research [75–79]. Understanding the trajectory of LLMs in SE will not only help identify their limitations but also uncover new applications and possibilities for transforming software development practices. In this section, we will explore the key future directions and research opportunities for LLMs in software engineering, ranging from technical advancements to ethical considerations and new ways to collaborate with AI models.

6.1. Specialization and domain-specific LLMs

A major area of research in the future will focus on creating more specialized LLMs tailored for specific domains within software engineering. While general-purpose LLMs like GPT-4 and Codex are highly effective across a wide range of coding tasks, they are often not optimized for niche areas such as embedded systems, real-time applications, or domain-specific languages like hardware description languages (HDL). Researchers are likely to focus on training LLMs on highly curated, domain-specific datasets, allowing these models to gain deeper expertise in specialized fields. For example, an LLM trained exclusively on medical software code or financial systems might be better equipped to understand the particular regulatory requirements, security needs, and performance constraints of these industries. Such domain-specific models could also include compliance checks that align with industry-specific standards, helping ensure that software adheres to legal and regulatory frameworks. Similarly, LLMs could be fine-tuned for particular programming languages or frameworks, providing deeper insights and optimizations tailored to those specific environments.

6.2. Improved interpretability and explainability

One of the most pressing challenges with the current generation of LLMs is their "black-box" nature, meaning they often provide answers or code suggestions without clear explanations of how or why those suggestions were made. This lack of transparency is problematic, particularly in safety-critical applications like healthcare, finance, or aerospace, where understanding the reasoning behind code is essential for ensuring security and correctness. Research in this area will likely focus on improving the interpretability and explainability of LLMs. Efforts will be made to create models that can not only generate code but also explain the rationale behind their decisions, offering developers more confidence in the accuracy and safety of the suggestions. This could involve developing new methods for LLMs to highlight the key parts of the training data or coding patterns that influenced their output. Explainable AI (XAI) frameworks that allow for deeper interrogation of LLM outputs could become more commonplace in SE environments, helping engineers better understand the suggestions provided by the models.

6.3. Collaborative human-ai programming environments

The future of LLMs in software engineering will likely emphasize collaborative programming environments where humans and AI work together seamlessly. This will involve creating tools and platforms that promote symbiotic relationships between developers and LLMs, allowing both parties to complement each other's strengths. For instance, while LLMs excel at generating code quickly and efficiently, human developers bring contextual understanding, creativity, and ethical judgment that AI currently lacks. Research opportunities in this area include developing more intuitive, conversational interfaces for LLMs, where developers can interact with models in a fluid and iterative manner. This could involve advancements in multimodal AI, where LLMs can take into account visual inputs, such as system diagrams or wireframes, to better understand the developer's intent and provide more relevant suggestions. Similarly, AI models could be trained to adapt their suggestions based on real-time feedback from developers, improving their effectiveness over time and enabling a more interactive coding process. These collaborative environments could also include AI models acting as "pair programmers," offering continuous feedback, alternative coding approaches, and potential optimizations during the development process.

6.4. Enhanced debugging and automated bug fixing

One of the most promising future directions for LLMs in software engineering is their potential to revolutionize debugging and automated bug fixing. Current LLMs can already identify and suggest solutions for common errors, but future advancements may lead to more sophisticated debugging tools that can understand complex bugs in large, multi-component systems. Future research may focus on training LLMs to detect not just surface-level issues (e.g., syntax errors), but deep-rooted logical bugs, performance bottlenecks, and security vulnerabilities in more extensive codebases. For instance, LLMs of the future could autonomously analyze code dependencies and execution paths to identify the root cause of subtle issues, such as memory leaks or race conditions, which are difficult to detect manually. Moreover, they could propose multiple solutions, weigh the pros and cons of each, and recommend the best course of action, tailored to specific system constraints. Further research could explore the potential for AI to continuously monitor running systems and automatically suggest patches or improvements in real-time, reducing the need for human intervention in maintenance tasks.

6.5. Ethical and security concerns

As LLMs become more prevalent in SE, the ethical and security implications of their use will require ongoing research. For instance, as LLMs generate more and more code, questions about the ownership and licensing of that code will arise, particularly when the models are trained on publicly available, open-source projects. Who owns the code generated by AI models, and how do we ensure that it complies with existing intellectual property laws? Addressing these issues will require interdisciplinary research that involves not just software engineering but also legal scholars, ethicists, and policy makers. Another major area of concern is the security of AI-generated code. Although LLMs can detect certain types of vulnerabilities, they can also inadvertently introduce new ones. Research will need to focus on creating mechanisms that prevent LLMs from generating insecure code, particularly in mission-critical systems. There is also the risk of bias and ethical dilemmas in the datasets used to train LLMs. Models trained on biased or incomplete data may perpetuate harmful stereotypes or make inaccurate decisions, which could have significant consequences in sectors such as healthcare or criminal justice software systems. Future research will need to address ways to mitigate these risks, ensuring fairness and accountability in AI-generated code.

6.6. Continual learning and model adaptation

As software development environments evolve, so too must the LLMs that support them. One area of research is continual learning; where LLMs can update their knowledge in real-time as they are exposed to new coding patterns, languages, or technologies. This would eliminate the need for retraining models from scratch and allow LLMs to stay relevant in dynamic environments. Future LLMs could potentially learn from real-world codebases as they evolve, adapting to new trends in development practices and adjusting their suggestions accordingly. Moreover, research into adaptive LLMs may explore models that can fine-tune themselves based on specific user needs or project contexts. For instance, a developer working on a web application might receive different types of suggestions from an LLM compared to someone working on an embedded system. Models could be fine-tuned not just for specific industries but also for individual developers, offering personalized feedback based on past interactions, coding styles, and preferred development frameworks.

6.7. Cross-Language and multimodal development

With the rise of LLMs in software engineering, there is growing interest in models that can understand and generate code across multiple programming languages. This capability would be especially useful for projects that involve integrating systems built in different languages or for teams with diverse language preferences. Research opportunities in this area include developing LLMs that are fluent in cross-language development, offering seamless transitions between languages and ensuring that code components written in different languages can work together efficiently. Additionally, multimodal LLMs that can integrate text, code, and even visual information (such as UI wireframes or architectural diagrams) offer exciting possibilities for the future of SE. These models could enable more comprehensive understanding of complex software systems, allowing developers to describe features in natural language while the LLM generates code, suggests optimizations, and aligns it with the visual or architectural elements of the project.

6.8. Education and training in the AI era

Lastly, the rise of LLMs in software engineering will have a profound impact on how future developers are trained and educated. As LLMs take over more of the rote coding tasks, the focus of SE education will likely shift toward higher-level problem-solving, system design, and ethical decision-making. Researchers will explore new pedagogical models that emphasize the collaboration between humans and AI, teaching developers not only how to code but also how to work effectively with AI tools. Future research in education will likely investigate how to integrate LLMs into software engineering curricula, ensuring that developers are well-prepared to work with AI-enhanced development tools. There will also be a need to develop new metrics for assessing coding skills, as the traditional focus on syntax and manual coding proficiency may become less relevant in a world where LLMs handle much of the low-level programming work.

7. Conclusion

The integration of LLMs into software engineering represents a significant turning point in how software is developed, maintained, and optimized. This article has explored the potential of LLMs to both enhance and challenge the current practices within the field of software engineering. Throughout the discussion, several important findings have emerged regarding the use of LLMs. They have proven to be game-changers across various phases of the software development lifecycle, including requirement analysis, code generation, testing, and debugging. By automating routine tasks and improving code quality, LLMs allow developers to focus on more complex and creative aspects of their work. Furthermore, LLMs can ensure consistency across large codebases and assist in maintaining legacy systems, thereby addressing technical debt effectively. However, while the potential of LLMs is vast, ethical concerns surrounding data bias, intellectual property, and job displacement must be carefully managed. The computational costs associated with training and deploying these large-scale models can also be prohibitive, particularly for smaller organizations. In light of these findings, it is clear that LLMs are not merely a product of overhyped marketing; they represent a profound shift in how software is engineered. They should be seen as powerful tools that augment human capabilities rather than replace them. Human oversight remains crucial for ensuring that AI-generated code aligns with project goals, is secure, and is free from biases. Therefore, the verdict is that LLMs indeed are game-changers in software engineering, but their true potential can only be unlocked when combined with human expertise and ethical safeguards. For developers and organizations, embracing the rise of LLMs is not just a choice but a strategic imperative. Developers must become familiar with how LLMs can assist in coding, testing, debugging, and maintenance while continuing to refine their higher-level skills such as system design and ethical decision-making. Organizations should invest in integrating LLMs into their development environments, starting with pilot projects to gauge effectiveness, as this can reduce development costs, accelerate time-to-market, and enhance software quality. Educational institutions, too, should revise their software engineering curricula to prepare the next generation of developers for the future of AI-driven development.

The impact of this article extends beyond simply presenting the advantages and challenges of LLMs in software engineering; it provides a balanced and nuanced perspective that allows stakeholders to make informed decisions about adopting these technologies. By highlighting real-world case studies, technical strengths, ethical considerations, and future research opportunities, the article contributes to the growing discourse on AI-driven development tools and their place in the future of software engineering. Ultimately, it serves as a guide for developers, organizations, and researchers, helping them understand how LLMs can enhance workflows and the skills needed to remain competitive in an AI-driven landscape. As LLMs continue to evolve, their integration into software engineering practices will redefine what is possible in software development, pushing the boundaries of automation, creativity, and collaboration. Thus, this article offers a foundational understanding of how LLMs are poised to change the software engineering landscape, encouraging stakeholders to embrace these tools thoughtfully and strategically. In conclusion, LLMs hold the potential to significantly disrupt and enhance the software engineering process, and as developers and organizations adapt to these changes, they will find themselves at the forefront of a new era in software development—one that is faster, more efficient, and more collaborative than ever before.

Funding

The authors declare that no funds, grants, or other supports were received during the preparation of this manuscript.

Data availability

No datasets were generated or analyzed during the current study.

CRediT authorship contribution statement

Md. Asraful Haque: Writing – original draft, Methodology, Investigation, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financialinterestsor personal relationships that could have appeared to influence the work reported in this paper.

References

[1] Guide to the Software Engineering Body of Knowledge (SWEBOK Guide), in: H. Washizaki (Ed.), Guide to the Software Engineering Body of Knowledge (SWEBOK Guide), IEEE Computer Society, 2024.
[2] R.S. Pressman. Software Engineering: A Practitioner's Approach. 5th Edition, McGraw-Hill Higher Education, 2001, ISBN- 0073655783.
[3] P. Jalote. Software Engineering: A Precise Approach. Wiley, 2010, ISBN-9788126523115.
[4] M.A. Haque, N. Ahmad, Key issues in software reliability growth models, Recent Adv. Comput. Sci. Commun. 15 (5) (2022) 741–747.
[5] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, Li Li, X. Luo, D. Lo, J. Grundy, H. Wang, Large language Models for Software engineering: a systematic literature review, ACM Trans. Soft. Eng. Methodol. (2024), https://doi.org/10.1145/3695988.
[6] I. Ozkaya, Application of large language models to software engineering tasks: opportunities, risks, and implications, IEEE Softw. 40 (3) (2023) 4–8.
[7] S. Rasnayaka, G. Wang, R. Shariffdeen, G.N. Iyer, An empirical study on usage and perceptions of LLMs in a software engineering project, in: Proceedings of the 1st International Workshop on Large Language Models for Code, 2024, pp. 111–118, https://doi.org/10.1145/3643795.3648379. Pages.
[8] Y. Li, T. Zhang, X. Luo, H. Cai, S. Fang, D. Yuan, Do pretrained language models indeed understand software engineering tasks? IEEE Trans. Software Eng. 49 (10) (2023) 4639–4655.
[9] Z. Liu, Y. Tang, X. Luo, Y. Zhou, L.F. Zhang, No need to lift a finger anymore? Assessing the quality of code generation by ChatGPT, IEEE Trans. Software Eng. 50 (6) (2024) 1548–1584.
[10] A. Tarassow. 2023. The potential of llms for coding with low-resource and domain-specific programming languages. arXiv preprint arXiv:2307.13018.
[11] J. Sallou, T. Durieux, A. Panichella, Breaking the silence: the threats of using LLMs in software engineering, in: Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results, 2024, pp. 102–106. Pages.
[12] A. Vaswani, et al., Attention Is All You Need, 30, Advances in Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA, 2017.
[13] T.B. Brown, et al., Language models are few-shot learners, in: Proc. of the 34th Int. Conf. on Neural Information Processing Systems (NIPS '20), 2020, pp. 1877–1901. Article 159.
[14] J. Howard, S. Ruder, Universal language model fine-tuning for text classification, in: 56th Annual Meeting of the Association for Computational Linguistics (Long Papers), 2018, pp. 328–339, pagesMelbourne, Australia.
[15] L. Ouyang, et al., Training language models to follow instructions with human feedback, in: 36th Conf. on Neural Information Processing Systems 35, 2022, pp. 27730–27744.
[16] M.A. Haque, S. Li. 2024. Exploring ChatGPT and its impact on society. AI and ethics. doi:10.1007/s43681-024-00435-4.
[17] M. Lubbad. 2023. GPT-4 parameters: unlimited guide NLP's game-changer. Medium (March 19, 2023). Available online: https://medium.com/@mlubbad/the-ultimate-guide-to-gpt-4-parameters-everything-you-need-to-know-about-nlps-game-changer-109b8767855a.
[18] W.X. Zhao et al. 2023. A survey of large language models. arXiv preprint arXiv: 2303.18223v11.
[19] R. Taylor, M. Kardas, G. Cucurull, T. Scialom, A. Hartshorn, E. Saravia, A. Poulton, V. Kerkez, R. Stojnic. 2022. Galactica: a large language model for science. arXiv preprint arXiv:2211.09085.
[20] H. Touvron et al. 2023. Llama: open and efficient foundation language models. arXiv preprint arXiv:2302.13971.
[21] M.A. Haque, A brief analysis of 'ChatGPT' – A revolutionary tool designed by OpenAI, EAI Endorsed Tran. AI and Robotics 1 (1) (2023) e15.
[22] Y. Chang, et al., A survey on evaluation of large language models, ACM Trans. Intell. Syst. Technol. 15 (3) (2024) 1–45. VolIssueArticle No. 39Pages.
[23] Jiang, J., et al. 2024. A survey on large language models for code generation. arXiv preprint arXiv:2406.00515.
[24] S. Zhang, J. Wang, G. Dong, J. Sun, Y. Zhang, G. Pu. 2024. Experimenting a new programming practice with llms. arXiv preprint arXiv:2401.01062.
[25] R.A. Husein, H. Aburajouh, C. Catal, Large language models for code completion: a systematic literature review, Comput. Stand. Interf. 92 (2025) 103917.
[26] M. Welsh, The end of programming, Commun. ACM 66 (1) (2023) 34–35, https://doi.org/10.1145/3570220.
[27] Z. Wang, J. Li, Ge Li, Z. Jin. 2023. Chatcoder: chat-based refine requirement improves llms' code generation. arXiv preprint arXiv:2311.00272.
[28] Z. Yu et al. 2024. HumanEval Pro and MBPP Pro: evaluating large language models on self-invoking code generation. arXiv preprint arXiv:2412.21199v2.
[29] K. Huang, et al., An empirical study on fine-tuning large language models of code for automated program repair, in: Proceedings 38th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE), 2023, pp. 1162–1174.
[30] M.A. Haque, S. Li, The potential use of ChatGPT for debugging and bug fixing, EAI Endorsed Trans. AI and Robot. 2 (1) (2023) e4.
[31] S. Kang, J. Yoon, N. Askarbekkyzy, S. Yoo, Evaluating diverse large language models for automatic and general bug reproduction, IEEE Trans. Software Eng. 50 (10) (2024) 2677–2694.
[32] Z. Fan, X. Gao, M. Mirchev, A. Roychoudhury, S. Hwei Tan, Automated repair of programs from large language models, in: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), Melbourne, Australia, 2023, pp. 1469–1481.

[33] M. Schafer, S. Nadi, A. Eghbali, F. Tip, An empirical evaluation of using large language models for automated unit test generation, IEEE Trans. Software Eng. 50 (1) (2024) 85–105.

[34] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, Q. Wang, Software testing with large language models: survey, landscape, and vision, IEEE Trans. Software Eng. 50 (4) (2024) 911–936.

[35] Y. Wang et al. 2025. ProjectTest: a project-level LLM unit test generation benchmark and impact of error fixing mechanisms. arXiv preprint arXiv: 2502.06556v4.

[36] J.A. Pizzorno and E.D. Berger. 2025. CoverUp: coverage-guided LLM-based test generation. arXiv preprint arXiv:2403.16218v3.

[37] T. Xue, X. Li, T. Azim, R. Smirnov, J. Yu, A. Sadrieh, B. Pahlavan. 2024. Multi-programming language ensemble for code generation in large language model. arXiv preprint arXiv:2409.04114.

[38] J. Zhang, P. Nie, J.J. Li, M. Gligoric, Multilingual code Co-evolution using large language models, in: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2023), 2023, pp. 695–707.

[39] Q. Peng, Y. Chai, X. Li, Humaneval-xl: a multilingual code generation benchmark for cross-lingual natural language generalization, in: Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation, 2024, pp. 8383–8394, pagesTorino, Italia.

[40] J. Xing, M. Bhatia, S. Phulwani, D. Suresh, R. Matta. 2025. HackerRank-ASTRA: evaluating correctness & consistency of large language models on cross-domain multi-file project problems. arXiv preprint arXiv:2502.00226v1.

[41] A. Shirafuji, Y. Oda, J. Suzuki, M. Morishita, Y. Watanobe, in: Refactoring Programs Using Large Language Models with Few-Shot Examples. 30th Asia-Pacific Software Engineering Conference (APSEC-23), Seoul, South Korea, 2023, pp. 151–160.

[42] S. Ishida et al. 2024. LangProp: a code optimization framework using large Language Models applied to driving. arXiv preprint arXiv:2401.10314.

[43] P. Akioyamen. 2024. The unreasonable effectiveness of LLMs for query optimization. arXiv preprint arXiv:2411.02862v1.

[44] Z. Yao et al. 2025. A query optimization method utilizing large language models. arXiv preprint arXiv:2503.06902v1.

[45] A.D. Porta, et al., Using large language models to support software engineering documentation in waterfall life cycles: are we there yet?, in: Proceedings of the 4th National Conference on Artificial Intelligence, organized by CINI, May 29-30, Naples, Italy, 2024.

[46] L. Belzner, T. Gabor, M. Wirsing, Large language model assisted software engineering: prospects, challenges, and a case study, in: Bridging the Gap Between AI and Reality: First International Conference, AISoLA 2023, Crete, Greece, 2023, pp. 355–374.

[47] H. Jin, L. Huang, H. Cai, J. Yan, Bo Li, H. Chen. 2024. From LLMs to LLM-based Agents for Software Engineering: a survey of current, challenges and future. arXiv preprint arXiv:2408.02479.

[48] F. Errica, G. Siracusano, D. Sanvito, R. Bifulco. 2024. What did I do wrong? Quantifying LLMs' Sensitivity and consistency to prompt engineering. arXiv preprint arXiv:2406.12334v1.

[49] Y. Xia, J. Kim, Y. Chen, H. Ye, S. Kundu, C. Hao, N. Talati. 2024. Understanding the performance and estimating the cost of LLM fine-tuning. arXiv preprint arXiv: 2408.04693.

[50] M.C. Rillig, M. Agerstrand, M. Bi, K.A. Gould, U. Sauerland, Risks and benefits of large language models for the environment, Environ. Sci. Technol. 57 (9) (2023) 3464–3466.

[51] C. Tantithamthavorn, J. Cito, H. Hemati, S. Chandra, Explainable AI for SE: experts' interviews, challenges, and future directions, IEEE Softw. 40 (4) (2023).

[52] B. Kou, S. Chen, Z. Wang, L. Ma, and T. Zhang. 2023. Do large language models pay similar attention like Human programmers when generating code?. arXiv preprint arXiv:2306.01220.

[53] N. Perry, M. Srivastava, D. Kumar, D. Boneh, Do users write more insecure code with AI assistants?, in: ACM SIGSAC Conference on Computer and Communications Security (CCS '23), 2023, pp. 2785–2799.

[54] C. Kirchhubel, G. Brown, Intellectual property rights at the training, development and generation stages of Large language Models, in: Proceedings of the Workshop on Legal and Ethical Issues in Human Language Technologies @ LREC-COLING 2024, ELRA and ICCL, Torino, Italia, 2024, pp. 13–18, pages.

[55] J. Jiao, S. Afroogh, Y. Xu, C. Phillips. 2024. Navigating LLM Ethics: advancements, challenges, and future directions. arXiv preprint arXiv:2406.18841.

[56] S. Imai, Is GitHub copilot a substitute for human pair-programming? An empirical study, in: Proceedings of the ACM/IEEE 44th International Conference on Software Engineering (ICSE '22), Pittsburgh, Pennsylvania, 2022, pp. 319–321.

[57] M. Jaworski, D. Piotrkowski. 2023. Study of software developers' experience using the Github Copilot Tool in the software development process. arXiv preprint arXiv: 2301.04991.

[58] S. Peng, E. Kalliamvakou, P. Cihon, M. Demirer. 2023. The impact of AI on developer productivity: evidence from GitHub copilot. arXiv preprint arXiv: 2302.06590.

[59] D. Smit, H. Smuts, P. Louw, J.;. Pielmeier, C. Eidelloth, The impact of GitHub Copilot on developer productivity from a software engineering body of knowledge perspective, in: AMCIS 2024 Proceedings, 2024, p. 10.

[60] A. Ziegler, E. Kalliamvakou, X.A. Li, A. Rice, D. Rifkin, S. Simister, G. Sittampalam, E. Aftandilian, Measuring GitHub copilot's impact on productivity, Commun. ACM 67 (2024) 54–63, 3 (March 2024).

[61] R. Salva. 2025. Essentials of GitHub copilot. Available online: https://resources.github.com/learn/pathways/copilot/essentials/essentials-of-github-copilot/.

[62] Revolutionizing Shopify's Engineering: The Power of GitHub Copilot, Available online, https://www.toolify.ai/ai-news/revolutionizing-shopifys-engineering-the-power-of-github-copilot-821646, 2024.

[63] A. Ambo. Empowering devs with AI: how Shopify made GitHub Copilot core to its culture. 2023. Available online: https://medium.com/@AmboAtsushi/empowering-devs-with-ai-how-shopify-made-github-copilot-core-to-its-culture-d237ba09d61b.

[64] Impactful work: helping developers around the world improve productivity with AI. 2023. Available online: https://aws.amazon.com/careers/life-at-aws-impactful-work-helping-developers-around-the-world-improve-productivity/.

[65] V.K. Sikha, AI fueled transformation in application development & coding, Int. J. Commun. Networks Inf. Security 15 (04) (2023).

[66] V. Joshi, I. Band. 2024. Disrupting test development with AI assistants. arXiv preprint arXiv:2411.02328.

[67] Tabnine: using Google Cloud powered AI to help developers code faster. 2023. Available online: https://cloud.google.com/customers/tabnine.

[68] K.-.A. Marvel. 2025. Codeium: the best github copilot alternative. Available online: https://semaphore.io/blog/codeium.

[69] Clearwater Analytics on Codeium, Available online, https://codeium.com/blog/clearwater-analytics-case-study, 2024.

[70] A. Masood, S.Dahal, A. Cai, G. Burtini. 2022. Ghostwriter AI & Complete Code Beta. Replit Blog: https://blog.replit.com/ai.

[71] A. Masad. 2023. Replit's path to product-market fit — The $1 billion side project. Available online: https://review.firstround.com/replits-path-to-product-market-fit/.

[72] Cody + Leidos: maximizing efficiency with heightened security in the AI race. 2025. Available online: https://sourcegraph.com/case-studies/cody-leidos-maximizing-efficiency-heightened-security-ai-race.

[73] N. Maslej, L. Fattorini, R. Perrault, V. Parli, A. Reuel, E. Brynjolfsson, J. Etchemendy, K. Ligett, T. Lyons, J. Manyika, J.C. Niebles, Y. Shoham, R. Wald, J. Clark, The AI Index 2024 Annual Report, in: AI Index Steering Committee, Institute for Human-Centered AI, Stanford University, CA, 2024.

[74] Kyle Daigle & GitHub Staff. 2024. Survey: the AI wave continues to grow on software development teams. GitHub blog. Available online: https://github.blog/news-insights/research/survey-ai-wave-grows/#key-survey-findings.

[75] D. Russo, Navigating the complexity of generative AI adoption in software engineering, ACM Trans. Softw. Eng. Methodol. 33 (2024) 50, https://doi.org/10.1145/3652154, 5, Article 135 (June 2024)pages.

[76] A. Drake. 2024. The future of software engineering: lLMs and beyond. Comet (February 28, 2024). Available online: https://www.comet.com/site/blog/the-future-of-software-engineering-llms-and-beyond/.

[77] J. Monti. 2024. The future of AI-driven software development. Medium (Mar 8, 2024). Available online: https://joemonti.org/the-future-of-ai-driven-software-development-0dec24759a71.

[78] J. Sauvola, S. Tarkoma, M. Klemettinen, J. Riekki, D. Doermann, Future of software development with generative AI, Automat. Software Eng. 31 (2024) 26, https://doi.org/10.1007/s10515-024-00426-z.

[79] V. Terragni, P. Roop, K. Blincoe. 2024. The future of software engineering in an AI-driven world. arXiv preprint arXiv:2406.07737.