Not All Problems Are Nails, Not All Tools Should Be Hammers: A Position Paper on Agent Usage in Software Engineering Tasks

Juuso Rytilahti
University of Turku
Turku, Finland
jubery@utu.fi

Panu Puhtila
University of Turku
Turku, Finland
papuht@utu.fi

Oshani Weerakoon
University of Turku
Turku, Finland
osweer@utu.fi

Erkki Kaila
University of Turku
Turku, Finland
ertaka@utu.fi

Tuomas Mäkilä
University of Turku
Turku, Finland
tusuma@utu.fi

Abstract

The use of AI-powered tools in software engineering (SWE) has increased significantly, primarily due to advancements in large language models (LLMs). LLMs can generate code from natural language prompts and even produce complete software artifacts. Along with these changes comes a new class of people working in the software industry: citizen developers. Citizen developers generally have no technical background but can produce technical applications or artifacts with the aid of LLMs. Moreover, LLM-powered AI agents are being used across many application areas, and there is a trend towards using such agents to solve all problems. These changes merit consideration of how, by whom, and when SWE tasks should be automated. Throughout this paper, we argue that some problems should be solved with LLMs, while others should not. We point out that the developers' backgrounds matter as much as the problems to be solved in this regard. The perspectives we offer in this paper suggest that future research should consider the limitations of both the users' knowledge and the technology behind the tools.

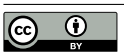
CCS Concepts

• **Human-centered computing** → **Human computer interaction (HCI)**; • **Software and its engineering** → **Software creation and management**; • **Language models**;

Keywords

Software Engineering, Large Language Models, Autonomous Agents, Citizen Developers

ACM Reference Format:
Juuso Rytilahti, Panu Puhtila, Oshani Weerakoon, Erkki Kaila, and Tuomas Mäkilä. 2026. Not All Problems Are Nails, Not All Tools Should Be Hammers: A Position Paper on Agent Usage in Software Engineering Tasks. In *International Workshop on Agentic Engineering (AGENT '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3786167.3788413



AGENT '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2399-5/26/04
https://doi.org/10.1145/3786167.3788413

1 Introduction

We are currently experiencing a moment where advances in artificial intelligence (AI), and especially in the fields of large language models (LLMs), could be called a new revolution in automation. Day by day, it seems that the LLM performance is in constant increase, with newer and better models being rolled out from the research and development divisions of technology giants.

In recent years, the development of new features such as support for multimodal inputs, retrieval-augmented generation (RAG), voice-controlled agents, and autonomous AI agents has improved the usability of LLMs. These innovations have also given rise to phenomena such as vibe coding [57], citizen developers, and low/code no code platforms. *Citizen developers* are defined as people who engage in software development without prior technical expertise [51]. This phenomenon has emerged from the proliferation of low-code/no-code platforms. These platforms offer easy-to-use editing interfaces that allow users to build simple applications by connecting ready-made components and simplified scripting languages [8, 55]. These new ways of interacting with computers democratize software product development, enabling individuals with limited or no technical expertise to explore their interests in this field.

It is becoming apparent that while AI technologies are useful in many tasks, they are not the best answer to all problems. For example, a recent MIT report [1] found that only 5% of custom enterprise AI tools and 40% of general-purpose LLMs were successfully implemented into production. In their report, a successful implementation was defined as one where the users or executives defined AI to have a measurable positive impact on productivity or on profit and loss (P&L) performance. This observation supports the hypothesis that, even with current advancements, AI technologies are not the perfect, one-size-fits-all solution they have been hyped up to be. This reflects the argument Etzioni made in 1996. [18]:

> *We need to recognize that intelligent agents are ninety-nine percent computer science and one percent AI.*

In this paper, we present considerations on the correct and meaningful use cases for deploying artificial intelligence agents, based on previous research. We also discuss how the background of the developer (citizen or technical) defines the usage of AI. Additionally, we reflect on the limitations of LLMs to offer insights into how

[1] https://mlq.ai/media/quarterly_decks/v0.1_State_of_AI_in_Business_2025_Report.pdf

their implementation could fail. Not all problems are nails, so not all solutions should be hammers.

First, we need to define what agentic AI actually is. Earliest theoretical definitions go back almost thirty years; In 1997, Wooldridge used the following definition to describe an intelligent agent [70]:

> *An intelligent agent is generally regarded as an autonomous decision-making system, which senses and acts in some environment.*

This describes modern LLM-powered agents quite accurately, albeit at a high level of abstraction. Based on these principles, the definition and practical implementation of agentic AI has evolved into its modern form. In this form, agents are understood to be software applications containing multiple independent LLM components that can communicate with each other and utilize external programs to achieve their goals, thus allowing varying degrees of autonomy. An agent's autonomy depends on its LLM components, their role in the architecture, and its prompting. Simple AI agents are not much more advanced than classical applications, while more complex agents can harness data, make decisions, and access external resources with minimal human supervision. [10, 26, 35]. In other words, AI agents are a mixture between classical sofware engineering techniques combined with the capabilities of LLMs, resulting in applications that can complete complex tasks with minimal human involvement.

This paper is structured as follows. First, we examine the limitations of large language models from various perspectives. Next, we discuss the quality of code produced by LLMs, its implications for software artifacts, and its effect on the software industry. Next, we discuss how developers' backgrounds and perspectives affect the use of AI in software development. Finally, we discuss our findings and conclude the paper.

2 Innate Limitations of LLMs

The idea of an automated agent is to perform autonomous, varied, complex, and often reasoning-related tasks [16, 45]. This, however, is not always in line with the capabilities of even the current state-of-the-art (SOTA) LLMs. Although LLMs can process information in different ways, they have limitations that prevent them from reaching their full potential.

2.1 Hallucinations and Biases

Perhaps the best-known type of LLM limitation is hallucinations, which means information generated by the model that is incorrect, nonsensical, or misaligned with the original query [28, 63]. Several studies [36, 39] have been conducted to understand the reasoning behind hallucination in LLMs, as well as ways to mitigate and evaluate them. Among the identified reasons are inference methods and model training types [36], as well as the use of spurious data [22]. Notable efforts to mitigate and benchmark hallucinations include the Hallucination Benchmark HaluEval 2.0 [36] and Factualness Evaluations via Weighting LLMs (FEWL) [68].

One of the downsides of the LLMs is their tendency to use in-context learning shortcuts, also known as *spurious correlations* [61, 73]. In short, spurious correlation occurs due to biases in LLM training data. A comprehensive picture of different shortcut phenomenon-related approaches can be seen, in e.g. [61], which encompasses, e.g., token bias, label bias, and position bias, and also classifies the mitigation approaches presented in various studies into three distinct types: data-centric, model-centric, and prompt-centric approaches. Another subtle problem is that the LLM output can be sycophantic; that is, it often contains excessive flattery and overly agrees with the user [41].

2.2 Prompt and Input

LLM performance depends not only on the specific model, but also on the *prompt* instructing its behavior. Even minor variations in the prompt can result in different outputs. This is why prompt engineering has emerged as a practice that aims to improve output quality by meticulously defining prompts [44]. The methods include, for example, introducing examples in the input (few-shot prompting) [9], or instructing the model to reason [67]. There have also been efforts to automatically enhance the given prompts, such as Promptbreeder [19], Llama prompt-ops [2], or OpenAI prompt optimizer for GPT-5[3].

Various studies have inspected the effect of the input length on the performance of the LLM in different tasks. The so-called needle-in-the-haystack benchmarks, where the model is tasked with finding a specific piece of information among the input [58, 60], have become a standard way to measure the in-context retrieval performance of different models [58]. Schuster et al. [58] tested 7 LLMs' ability to find a needle-in-the-haystack on complex reasoning tasks, and found that the model performance tended to decline when a longer context was introduced. Schi et al. [60] found that the semantic surroundings of the piece of information retrieved affect the model performance in in-context information retrieval, and that it has a greater impact on performance than input length. Technical report from Chroma [24], where they tested 18 different SOTA models, noted a performance decrease with context increase.

2.3 Measuring Performance

The easiest way to overcome the limitations of the model context window is to improve the quality of the input. One automated way to do this is through different types of RAG systems. Barnett et al. [5] defined 7 failure points they faced when they built 3 different RAG-based systems across 3 different domains. These were related to missing content or documents, not extracting documents relevant to the query, the wrong format, or incorrect specificity of the output, and incomplete answers. Furthermore, there exists speculation that, for example, embedding-based retrieval may have theoretical limits that may be impossible to overcome [69], thus affecting their scalability and usability.

LLLM performance is often measured using benchmarks. Although benchmarking is a well-established method in computer and software engineering, it is finicky when applied to LLMs. For example, changing minor things in multiple-choice questions can affect the performance of the model in the task greatly [3]. Furthermore, AI can sometimes find a loophole that leads the model to exploit imperfections in the benchmark design, resulting in a better score. For example, in SWE Bench Verified [30], the model has been

[2] https://github.com/meta-llama/prompt-ops
[3] https://platform.openai.com/chat/edit?optimize=true

able to look for direct fixes in the future commit history of the fixed issues[4], thus improving the performance on the benchmark illicitly.

Another way to inspect the model capabilities is to look into their capabilities in logical and mathematical reasoning. Jiang et al. [29] note that LLMs are not genuine reasoners and tend to rely on superficial patterns in the models' training data. Mirzadeh et al. [48] note that adding irrelevant clauses that appear to be semantically close can hinder the model's performance in mathematical reasoning tasks up to 65%. This also seems to be in line with the studies regarding the impact of input size on information retrieval noted earlier.

Finally, there are differences in how different models can specialize in different tasks. For example, larger models are harder to fine-tune than the smaller ones [62]. Shenfeld et al. [59] noticed that models utilizing reinforcement learning preserve prior knowledge better than models utilizing supervised fine-tuning. Additionally, there appears to be a limit on the scalability of the current models [31].

2.4 Emergent Security Threats

LLMs and AI agents have been shown to be a cause for multiple new security risks [33]. One obvious aspect of this is the various security vulnerabilities found in AI-generated code, observed by Tóth et al. [65] and Panichella [53]. There may also be hard-to-detect threats embedded in a foundational model during the training phase. These include the so-called *sleeper agents* [25], where the model is trained to depict some behaviour after a certain trigger is used, and *subminimal learning* where the model gains behavioral traits from semantically unrelated data, which can occur if the trained model shares the same base model with the teacher model [14].

Other security risks are more straightforward. For example, *prompt injection* is a technique where the LLM-powered systems are manipulated to do something they are not supposed to do by injecting malicious prompting into their workflow [46]. This kind of malicious prompting can be hidden inside the contents of websites and text files making it a real threat to the efficiency and security of LLM-based systems. What makes the prompt injection particularly insidious threat is, that it can be deployed in the documents in ways that are not readily apparent to human observer, for example, by using "invisible" i.e., white text. Lin et al. [37] have shown that some scientific researchers use this technique to manipulate the LLM-assisted peer review process, meriting the consideration of whether LLMs should be used in assisting peer-review process at all.

3 LLMs and Software Engineering

3.1 Software Quality

Utilizing LLM-produced code often leads to localized solutions, which is a problem. A white paper by GitClear analyzed 153 million changed lines of code, gathered between 2020 and 2023, and noted a notable increase in code lines that are being reverted or updated less than 2 weeks after being committed to the repository, and a reduction in code-reuse [23]. In other words, the LLM generates the same solution in a different form, cluttering the repository with unnecessary code and increasing technical debt.

With the help of modern, LLM-based solutions, software can be developed by nonprofessional citizen developers. These tools also speed up the development process for professionals, making the creation of working prototypes significantly faster. However, this can result in lower-quality code [49], as people with little or no experience may be unable to assess the quality of the code produced by LLMs. This can result in software that is not optimized or that contains significant security vulnerabilities [7].

One example of using an LLM in software engineering is using it for refactoring. Multiple benchmarks exist to measure an LLM's ability to perform this task. In an empirical study, Liu et al. examined 180 refactoring opportunities in Gemini and ChatGPT. They found that ChatGPT identified 176 refactoring possibilities, 63.6% of which were at the same level or better than those identified by human participants. [38], while Gemini suggested 137 solutions. However, the authors noted that both models suggested a small number of unsafe refactorings [38]. Pomian et al. noted that after applying a technique they developed, LLM was able to suggest similar refactoring as developers in 53.4% of the tested 1752 cases [54]. They also seem to be able to suggest design patterns to some extent, for example, ChatGPT-3.5 answered 56 design pattern questions, 93% of which with a good level of detail [42]. Liu et al. [40] explored LLm capability in the IT operations, with a benchmark that included 9,070 questions, and their findings highlight the need to consider two aspects, performance and robustness of the models in this task.

Yang et al. [72] studied the generation of unit tests with open-source LLMs, and came to the conclusion that hallucinations exhibited by the LLMs are one of the key problems, and that traditional methods of countering hallucinations such as RAG and Chain-of-Thought (CoT) seemed to be ineffective in this use case. Another study by Ouedraogo et al. [52] came to similar conclusions, although their observations on the usefulness of CoT did differ somewhat.

3.2 Implications on Autonomous Systems Producing Software Artifacts

The ecosystem of software development places its own obstacles to overcome. For example, Zimmermann et al. [78] note that the NPM ecosystem [5] has multiple potential single points of failure, and that many packages are dependent on vulnerable code due to a lack of maintenance. Miller et al. did a quantitative analysis of 28,100 widely-used NPM packages, and noted that 15% of them were abandoned within a 6-year observation window [47]. Moreover, similar problems are also in other package ecosystems. Alfadel et al. conducted an empirical study, inspecting 1,396 vulnerability reports affecting 698 packages in the Python ecosystem (PyPi), and noted similar problems [2]. Furthermore, only a small number of developers are willing to announce the deprecation of a package, and a significant number of open-source Python packages are in poor maintenance [77].

These observations raise two questions. First, how well can the LLMs identify and use suitable repositories? And second, can they automatically detect repositories containing malicious code?

[4] https://github.com/SWE-bench/SWE-bench/issues/465

[5] https://www.npmjs.com

Regarding the LLM's ability to find suitable repositories, we could mention framework EnvX [11], which achieved a 74.07% execution completion rate and 51.85% task pass rate performance on benchmark GitTaskBench [50]. Wang et al. [66] developed MalPacDetector, which achieved a false positive rate of 1.3% and a false negative rate of 7.5% in the detection of npm packages containing malicious code using a dataset consisting of 7,309 packages, of which 3,258 contained malicious code. Similarly, Zahan et al. [75] code review workflow achieved a near-perfect accuracy with a dataset from MalwareBench [74], which comprised a total of 5,115 npm packages, of which 2,180 packages were infected with malicious code.

Despite the promising results, the final answer may not be straightforward. A malicious actor can easily mislead the models through obfuscation or prompt injection, which could result in the use of an infected repository. For example, Wyss et al. evaluated LLMs' ability to detect malicious NPM package updates and demonstrated that mild code obfuscations allow adaptive adversaries to bypass LLM-based detection [71].

3.3 Effect on the Software Industry

As the internet has proliferated, it has become commonplace for software development professionals to turn to online tutorials and message boards for guidance. In this sense, using an LLM to provide development advice is the next logical step in the evolution of the SWE ecosystem. Similarly, over the past two centuries, we have gradually moved towards using natural language in programming, reducing specificity and adding layers of abstraction that distance developers from how machines work. From this perspective, the rise of code generation through prompting by LLMs is a logical progression. The end goal of using agentic AI professionally is simple: a human enters complex instructions, such as software requirements or perhaps only an idea or problem statement in natural language. From these inputs, the LLM creates the desired output: a complete, scalable software artifact. However, the current state of affairs is more complex..

From a professional point of view, we must first examine at what phase of the development process LLMs and agentic AIs should be used. One could argue that this depends on multiple perspectives. A preliminary study measuring brain activity during essay writing tasks indicates that using AI after the initial draft has been written by hand increases brain activity, while using it beforehand reduces it [34]. Furthermore, Alami et al. note that in code review, the LLM seems to disrupt the natural accountability [1], as an LLM cannot be held responsible for outcomes produced. Another problematic aspect is the potential decrease in overall developer quality. Relying too heavily on LLMs to write code can result in an incomplete understanding of the fundamental principles of software development [20].

One example of evaluating the effectiveness of the software developers is a study conducted by Becker et al. [6], where they measured the productivity of 16 developers, with an average work experience of 5 years, noting that allowing experienced open-source developers to access AI tools actually increased completion time by 19%. A study by Martinovic et al. [43] concluded that AI tools

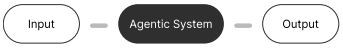


Figure 1: Presentation of how a non-technical person may be able only to assess the final output, and might view an agentic system essentially as a black box.

that aided in software development had increased the work satisfaction of the developers significantly, although in other measured factors, the increase in value was less visible. Additionally, Pomian et al. [54] created an IntelliJ IDEA [6] plugin for refactoring, and noted that 81.3% of 16 developers testing a plugin agreed with the recommendations made by the plugin.

For developers, LLMs and LLM-integrating tools can add an additional layer of quality assurance. One such example of applying LLMs in quality assurance is to study how LLMs are being used for code review tasks. Cihan et al. [13] study utilized the open-source Qodo PR Agent [7], and noted that the pull request closure time increased, but a more focused survey in 22 practitioners observed a minor improvement in code quality. One of the more recent works was conducted by Sun et al., who developed BitsAI-CR [64], which is a system that achieves high accuracy on industry-scale automating of code review with an automated, self-improving pipeline with 75% precision in review comment generation with over 12,000 weekly active users. Furthermore, similar results have been achieved in code refinement, utilizing a hybrid method that combined a rule-based system and LLMs, achieving up to 66% in code refinement generation and 79% accuracy in extracting reviewers' modification intent from the comments [21].

4 Perspective and Background Matters

Inspecting how people with differing backgrounds appraise the output and mid-steps of such tools or systems in the SWE context can provide an important perspective. As depicted in the Figure 1, a technically illiterate individual may see the system only as a black box, able to assess only the output of the system. On the other hand, an individual with a more profound technical background may understand the mechanisms of the system better, understanding all or at least some of the individual steps made to create the final output of the system. This is illustrated in Figure 2.

Thus, we should consider the perspectives of different groups, separated by their varying technical knowledge. This leads us to categorize the target groups into three distinct groups, each with varying technical knowledge: These groups include: 1) those with a proficient technical background, 2) those with a limited technical background, and 3) those without any technical background.

4.1 Individuals with strong technical backgrounds

Individuals with strong technical backgrounds can use AI agents to streamline mundane tasks and support different software development processes. Those individuals can also validate the output produced by these tools. Moreover, if they are familiar with the product or service under development, they can spot redundancies

[6] https://www.jetbrains.com/idea
[7] https://github.com/qodo-ai/pr-agent

Table 1: Common SWE use cases with justifications, and applicability by user type. Used abbreviations: individuals with strong technical knowledge (STK), individuals with moderate technical knowledge (MTK), citizen developers and other non-technical stakeholders (NTS), positive impact when used properly (++), slight positive impact when used properly (+), negative or no impact (-), and with major caveats (*).

Use case	Justification	STK	MTK	NTS
As a debugging tool.	The debugged problems are often common, trivial, and repetitive. Moreover, developers of all skill levels tend to become blind to their errors.	++	++	+*
As an additional quality assurance.	AI tends to repeat patterns in its training data, but still, domain knowledge is required.	++	++	-
As an additional source to find information of e.g., libraries.	Deep research tools can search the internet with a breathtaking scope. However, due to limitations of the current LLMs fueling agentic systems, utilizing these tools in information retrieval requires vigilance and domain knowledge.	++	+	-*
Code refactoring.	Research shows that LLMs won't catch all the refactoring opportunities, and thus, a professional dev with technical knowledge is needed to supplement AI's work. Moreover, there is a risk that AI refactorings break the code, emphasizing the need for technical knowledge.	++	++	+
Create a functional offline program.	As noted, there are fewer risks related to the development of an offline program with lacking knowledge and skills compared to software artifacts online.	+	++	+*
Creating prototypes or mock-ups.	Creating quick mock-ups helps to accelerate iteration and increase understanding of the problem space and of the potential solutions.	++	++	++
Helping to outline or define the problem statement.	Useful after initial rough sketching to help order and structure ideas. The research seems to indicate that AI should be used only after the initial sketches have been created.	+	+	+
Troubleshooting system configuration issues.	The risks relate to the user not understanding the effects of changes suggested by the agentic system.	++	+*	+*

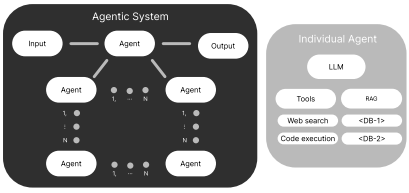


Figure 2: Presentation of how a technical person views the agentic system and its output (the AI-produced software artifact). In an ideal situation, they can understand, validate, and potentially fix the intermediate steps.

in the code because AI tends to focus on local solutions. Interestingly, professionals seem to adopt this new technology mainly based on how well it fits their existing practices and workflows instead of replacing them [56].

We could argue that professionals should have access to a more complex toolbox or set of features than citizen developers. Of course, individual skill levels vary. Some people may have technical understanding but lack the necessary skills. They may understand what kind of inputs are sufficient and have the ability to understand coding, but they lack the full comprehension that comes with training and learning the different skills required in SWE.

4.2 Citizen Developers and other non-technical stakeholders

Citizen developers, meaning individuals with little to no professional experience, are more likely to view AI-produced artifacts as essentially black boxes. This makes it difficult to assess the reliability of systems built by them. Given the current limitations of the technology discussed in this paper, we argue that these groups should primarily use autonomous agents for supportive processes in a professional setting. For instance, creating an initial draft of the desired software can help software professionals better understand the problem statement. The created prototype artifacts can be used to explain their strengths and weaknesses. Thus, the produced output can serve as a starting point for discussion.

These kinds of limitations may be important, as the non-technical stakeholders do not necessarily understand the potential pitfalls of technology. Not only are the potential security vulnerabilities an issue (e.g. Tóth et al. [65], Panichella [53]), but as the endlessly scaling cloud infrastructure has become more commonplace, the so-called *denial of wallet*[32] scenarios, also known as *cloud overflow*, are a real threat to organizational stability.

While this kind of scenario is usually the result of a cyberattack, similar outcomes are possible when code contains elements that can generate an unlimited amount of resources within a system without automated kill switches. The increased usage of AI-powered tools by citizen developers significantly increases these risks.

Nevertheless, it should be acknowledged that artifacts created by less professional users can be valuable in their own right. For example, Chow et al. [12] utilized vibe coding to create clinical teaching simulations, noting that such tools allow a shift in focus from technical constraints to pedagogical goals and enable fast prototyping. In Table 1 we have outlined common SWE use cases and related them to the knowledge and skill level of the person using the tool.

5 Problems worth solving

This leads us to the question: If systems orchestrated by autonomous agents are hammers, then what are the problems that these hammers should solve? We can look at a 1999 paper in which Jennings [27] notes that each successive development in software engineering is, generally,

> "*to make the engineering process easier or to extend the complexity of applications that can feasibly be built.*"

Jennings [27] noted that the methods developed up to that point fell short in three main ways:

> "*(i) the basic building blocks are too fine-grained; (ii) the interactions are too rigidly defined; or (iii) they possess insufficient mechanisms for dealing with organisational structure.*"

We can argue that agentic coding has solved these shortcomings at least partially. For example, Durrani et al. [17] show in their study how integration of AI in to SE industry has had a positive impact. However, as noted throughout this paper, the current progress has not come without its own set of problems.

In order to determine which problems these tools are meant to solve, we must first understand the limitations of the current models. Valid considerations include defining the problem statement, the end goal, and how easily the output can be verified. Additionally, assessing the likelihood of success, the rigidity of the final output or mid-step validation, and the resilience of the end product to mistakes can provide sound grounding. Ultimately, clarity is of importance, as the clarity of the problem definition, mid-steps, and solution dictates the likelihood of verifying success of AI in a given task.

Agents are constrained not only by AI capabilities but also by the limits of the external tools they use. Improving the LLM performance is not the only way to enhance the agent's overall performance. Other ways to improve the agent include giving it access to additional external tools or improving existing tools. Sometimes, the model or tool capability is sufficient, but the user must have background knowledge to validate the input provided by the agent to the tool. This can occur when, e.g., the agent's task involves identifying and using a statistical method to analyze a dataset. Thus the user experience of the agentic system can have a crucial impact on the system's usability and performance. For example, clearly separating the input and output of the tool from the LLM's output powering the agent can be helpful. Of course, sometimes the limitation is that the training data for the underlying models lacks sufficient examples, making the agent incapable of performing a task. In addition to this, LLMs are prone to spurious correlations and shortcut learning, which may lead to erroneous output.

As noted in Section 3.3, LLMs can be used as additional safeguards in quality assurance to identify problems in code that might be overlooked. Having this "second opinion" is useful since it is easy to become blind to the errors we make. In this way, LLMs can offer possible solutions, potentially saving significant amount of time.

We should also inspect the cost structure of software development. For example, Deghani and Ajrahimi separated costs related to software development into 2 different phases, production and maintenance, and noted that 90% of the costs during the software life are associated with its maintenance phase [15]. Zarnekov and Brenner surveyed costs over the application lifecycle on 30 IT application systems in three different companies for a production time of 5 years, and found that recurring costs for production and further development covered 79% of the overall costs [76]. A recent study by Ancan et al.[4] revealed that 55% of software developers already use LLM-assisted tools in maintenance tasks, and that 69% of these developers report an increase in productivity due to this. This suggests that LLM-based solutions can have a significant impact on the maintenance costs of software applications.

6 Conclusion

In this paper, we outlined when and where autonomous agents should be applied when automating different SWE tasks. For those without a technical background, autonomous agents should be used in well-defined, restricted problem spaces. Using them to produce software for direct production use without verifying the software security or functionality by a technically skilled individual is often not advisable. Agentic tools can help non-technical users outline the problem space and convey the necessary information to the developers, but also use no-code or low-code applications for development in environments that are restricted or secure in other ways.

More freedom can be allowed for persons with some background in SWE. The artifacts they produce may be used directly in the final software product, but direct use of such outputs should be carefully considered on a case-by-case basis. One way to mitigate risks is to allow only white-listed packages, libraries, or dependencies.

For those with a strong technical background, AI-based tools can provide a useful foundation. There is no need to define starting templates by hand anymore. Additionally, solutions can be adapted directly to the problem at hand to a certain extent. Minor errors can be detected and corrected. Furthermore, AI's tendency to follow patterns in its training data makes it ideal for ensuring that any commonly included aspects are not missed. Therefore, adding an AI-based tool for quality assurance, while not relying on it blindly, is recommended in most cases.

There is immense potential for automating different SWE tasks. However, despite constant performance increases, the models used by autonomous agents have innate limitations that hinder their usefulness. Current research focuses more on the capabilities than the limitations of these models. A core research initiative for future work is recognizing these shortcomings and paving the way for automating SWE tasks with these limitations in mind. Additionally, more emphasis should be placed on automating the maintenance of products instead of creating new software artifacts from scratch.

Original chatbot-focused interactions are the command-line tools of the modern era. Thus, integrating and designing different tools that utilize autonomous agents will change the landscape in the years to come, even if the development of the core of these tools—LLMs— would have already peaked.

Acknowledgments

This work has been supported by FAST, the Finnish Software Engineering Doctoral Research Network, funded by the Ministry of Education and Culture, Finland.

References

[1] Adam Alami, Victor Jensen, and Neil Ernst. 2025. Accountability in Code Review: The Role of Intrinsic Driversand the Impact of LLMs. *ACM Trans. Softw. Eng. Methodol.* 34, 8, Article 233 (Oct. 2025), 44 pages. doi:10.1145/3721127

[2] Mahmoud Alfadel, Diego Elias Costa, and Emad Shihab. 2023. Empirical analysis of security vulnerabilities in python packages. *Empirical Software Engineering* 28, 3 (2023), 59.

[3] Norah Alzahrani, Hisham Alyahya, Yazeed Alnumay, Sultan AlRashed, Shaykhah Alsubaie, Yousef Almushayqih, Faisal Mirza, Nouf Alotaibi, Nora Al-Twairesh, Areeb Alowisheq, M Saiful Bari, and Haidar Khan. 2024. When Benchmarks are Targets: Revealing the Sensitivity of Large Language Model Leaderboards. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 13787–13805. doi:10.18653/v1/2024.acl-long.744

[4] Oscar Ancán-Bastías, Carlos Cares, Jesús Peral, and Antonio Ferrández. 2026. AI in Software Maintenance: An Empirical Multi-source Approach. In *Research and Innovation Forum 2024*, Anna Visvizi, Orlando Troisi, Vincenzo Corvello, and Mara Grimaldi (Eds.). Springer Nature Switzerland, Cham, 101–111.

[5] Scott Barnett, Stefanus Kurniawan, Srikanth Thudumu, Zach Brannelly, and Mohamed Abdelrazek. 2024. Seven Failure Points When Engineering a Retrieval Augmented Generation System. In *Proceedings of the IEEE/ACM 3rd International Conference on AI Engineering - Software Engineering for AI* (Lisbon, Portugal) *(CAIN '24)*. Association for Computing Machinery, New York, NY, USA, 194–199. doi:10.1145/3644815.3644945

[6] Joel Becker, Nate Rush, Elizabeth Barnes, and David Rein. 2025. Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity. arXiv:2507.09089 [cs.AI] https://arxiv.org/abs/2507.09089

[7] Vladislav Belozerov, Peter J. Barclay, and Askhan Sami. 2026. LLMs in Coding and Their Impact on the Commercial Software Engineering Landscape. In *Advances in Computational Intelligence Systems*, Emma Hart, Tomas Horvath, Zhiyuan Tan, and Sarah Thomson (Eds.). Springer Nature Switzerland, Cham, 328–334.

[8] Alexander C Bock and Ulrich Frank. 2021. Low-code platform. *Business & Information Systems Engineering* 63, 6 (2021), 733–740.

[9] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[10] Chi-Min Chan, Weize Chen, Yusheng Su, Jianxuan Yu, Wei Xue, Shanghang Zhang, Jie Fu, and Zhiyuan Liu. 2023. ChatEval: Towards Better LLM-based Evaluators through Multi-Agent Debate. arXiv:2308.07201 [cs.CL] https://arxiv.org/abs/2308.07201

[11] Linyao Chen, Zimian Peng, Yingxuan Yang, Yikun Wang, Wenzheng Tom Tang, Hiroki H. Kobayashi, and Weinan Zhang. 2025. EnvX: Agentize Everything with Agentic AI. arXiv:2509.08088 [cs.AI] https://arxiv.org/abs/2509.08088

[12] Minyang Chow and Olivia Ng. 2025. From technology adopters to creators: Leveraging AI-assisted vibe coding to transform clinical teaching and learning. *Medical Teacher* 47, 12 (2025), 1927–1929. arXiv:https://doi.org/10.1080/0142159X.2025.2488353 doi:10.1080/0142159X.2025.2488353 PMID: 40202513.

[13] Umut Cihan, Vahid Haratian, Arda İçöz, Mert Kaan Gül, Ömercan Devran, Emircan Furkan Bayendur, Baykal Mehmet Uçar, and Eray Tüzün. 2025. Automated Code Review in Practice. In *2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 425–436. doi:10.1109/ICSE-SEIP66354.2025.00043

[14] Alex Cloud, Minh Le, James Chua, Jan Betley, Anna Sztyber-Betley, Jacob Hilton, Samuel Marks, and Owain Evans. 2025. Subliminal Learning: Language models transmit behavioral traits via hidden signals in data. arXiv:2507.14805 [cs.LG] https://arxiv.org/abs/2507.14805

[15] Sayed Mehdi Hejazi Dehaghani and Nafiseh Hajrahimi. 2013. Which factors affect software projects maintenance cost more? *Acta Informatica Medica* 21, 1 (2013), 63.

[16] Zane Durante, Qiuyuan Huang, Naoki Wake, Ran Gong, Jae Sung Park, Bidipta Sarkar, Rohan Taori, Yusuke Noda, Demetri Terzopoulos, Yejin Choi, Katsushi Ikeuchi, Hoi Vo, Li Fei-Fei, and Jianfeng Gao. 2024. Agent AI: Surveying the Horizons of Multimodal Interaction. arXiv:2401.03568 [cs.AI] https://arxiv.org/abs/2401.03568

[17] Usman Khan Durrani, Mustafa Akpinar, Muhammed Fatih Adak, Abdullah Talha Kabakus, Muhammed Maruf Öztürk, and Mohammed Saleh. 2024. A Decade of Progress: A Systematic Literature Review on the Integration of AI in Software Engineering Phases and Activities (2013-2023). *IEEE Access* 12 (2024), 171185–171204. doi:10.1109/ACCESS.2024.3488904

[18] Oren Etzioni. 1996. Moving up the information food chain: deploying softbots on the world wide web. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence - Volume 2* (Portland, Oregon) *(AAAI'96)*. AAAI Press, 1322–1326.

[19] Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. 2024. Promptbreeder: self-referential self-improvement via prompt evolution. In *Proceedings of the 41st International Conference on Machine Learning* (Vienna, Austria) *(ICML'24)*. JMLR.org, Article 541, 64 pages.

[20] Diana Franklin, Paul Denny, David A. Gonzalez-Maldonado, and Minh Tran. 2025. *Generative AI in Computer Science Education: Challenges and Opportunities*. Cambridge University Press.

[21] Qi Guo, Xiaofei Xie, Shangqing Liu, Ming Hu, Xiaohong Li, and Lei Bu. 2025. Intention is All you Need: Refining your Code from your Intention . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, USA, 1127–1139. doi:10.1109/ICSE55347.2025.00191

[22] Tianyang Han, Qing Lian, Rui Pan, Renjie Pi, Jipeng Zhang, Shizhe Diao, Yong Lin, and Tong Zhang. 2024. The Instinctive Bias: Spurious Images lead to Illusion in MLLMs. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (Eds.). Association for Computational Linguistics, Miami, Florida, USA, 16163–16177. doi:10.18653/v1/2024.emnlp-main.904

[23] William Harding and Matthew Kloster. 2024. *Coding on Copilot: 2023 Data Shows Downward Pressure on Code Quality*. Technical Report GitClear Report. GitClear / Alloy.dev Research. https://www.gitclear.com/coding_on_copilot_data_shows_ais_downward_pressure_on_code_quality Analyzed 153 million changed lines of code from January 2020 through December 2023.

[24] Kelly Hong, Anton Troynikov, and Jeff Huber. 2025. *Context Rot: How Increasing Input Tokens Impacts LLM Performance*. Technical Report –. Chroma Research. https://research.trychroma.com/context-rot Report published July 14, 2025.

[25] Evan Hubinger, Carson Denison, Jesse Mu, Mike Lambert, Meg Tong, Monte MacDiarmid, Tamera Lanham, Daniel M. Ziegler, Tim Maxwell, Newton Cheng, Adam Jermyn, Amanda Askell, Ansh Radhakrishnan, Cem Anil, David Duvenaud, Deep Ganguli, Fazl Barez, Jack Clark, Kamal Ndousse, Kshitij Sachan, Michael Sellitto, Mrinank Sharma, Nova DasSarma, Roger Grosse, Shauna Kravec, Yuntao Bai, Zachary Witten, Marina Favaro, Jan Brauner, Holden Karnofsky, Paul Christiano, Samuel R. Bowman, Logan Graham, Jared Kaplan, Sören Mindermann, Ryan Greenblatt, Buck Shlegeris, Nicholas Schiefer, and Ethan Perez. 2024. Sleeper Agents: Training Deceptive LLMs that Persist Through Safety Training. arXiv:2401.05566 [cs.CR] https://arxiv.org/abs/2401.05566

[26] Laurie Hughes, Yogesh K. Dwivedi, Tegwen Malik, Mazen Shawosh, Mousa Ahmed Albashrawi, Il Jeon, Vincent Dutot, Mandanna Appanderanda, Tom Crick, Rahul De', Mark Fenwick, Senali Madugoda Gunaratnege, Paulius Jurcys, Arpan Kumar Kar, Nir Kshetri, Keyao Li, Sashah Mutasa, Spyridon Samothrakis, Michael Wade, and Paul Walton. 2025. AI Agents and Agentic Systems: A Multi-Expert Analysis. *Journal of Computer Information Systems* 65, 4 (2025), 489–517. arXiv:https://doi.org/10.1080/08874417.2025.2483832 doi:10.1080/08874417.2025.2483832

[27] Nicholas R. Jennings. 1999. Agent-Oriented Software Engineering. In *Multiple Approaches to Intelligent Systems*, Ibrahim Imam, Yves Kodratoff, Ayman El-Dessouki, and Moonis Ali (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 4–10.

[28] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. 2023. Survey of Hallucination in Natural Language Generation. *ACM Comput. Surv.* 55, 12, Article 248 (March 2023), 38 pages. doi:10.1145/3571730

[29] Bowen Jiang, Yangxinyu Xie, Zhuoqun Hao, Xiaomeng Wang, Tanwi Mallick, Weijie J Su, Camillo Jose Taylor, and Dan Roth. 2024. A Peek into Token Bias: Large Language Models Are Not Yet Genuine Reasoners. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (Eds.). Association for Computational Linguistics, Miami, Florida, USA, 4722–4756. doi:10.18653/v1/2024.emnlp-main.272

[30] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? arXiv:2310.06770 [cs.CL] https://arxiv.org/abs/2310.06770

[31] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling Laws for Neural Language Models. arXiv:2001.08361 [cs.LG] https://arxiv.org/abs/2001.08361

[32] Daniel Kelly, Frank G. Glavin, and Enda Barrett. 2021. Denial of wallet—Defining a looming threat to serverless computing. *Journal of Information Security and Applications* 60 (2021), 102843. doi:10.1016/j.jisa.2021.102843

[33] Dezhang Kong, Shi Lin, Zhenhua Xu, Zhebo Wang, Minghao Li, Yufeng Li, Yilun Zhang, Hujin Peng, Zeyang Sha, Yuyuan Li, et al. 2025. A survey of llm-driven ai agent communication: Protocols, security risks, and defense countermeasures.

[34] Nataliya Kosmyna, Eugene Hauptmann, Ye Tong Yuan, Jessica Situ, Xian-Hao Liao, Ashly Vivian Beresnitzky, Iris Braunstein, and Pattie Maes. 2025. Your brain on ChatGPT: Accumulation of cognitive debt when using an AI assistant for essay writing task.

[35] Naveen Krishnan. 2025. AI Agents: Evolution, Architecture, and Real-World Applications. arXiv:2503.12687 [cs.AI] https://arxiv.org/abs/2503.12687

[36] Junyi Li, Jie Chen, Ruiyang Ren, Xiaoxue Cheng, Xin Zhao, Jian-Yun Nie, and Ji-Rong Wen. 2024. The Dawn After the Dark: An Empirical Study on Factuality Hallucination in Large Language Models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 10879–10899. doi:10.18653/v1/2024.acl-long.586

[37] Zhicheng Lin. 2025. Hidden Prompts in Manuscripts Exploit AI-Assisted Peer Review. arXiv:2507.06185 [cs.CY] https://arxiv.org/abs/2507.06185

[38] Bo Liu, Yanjie Jiang, Yuxia Zhang, Nan Niu, Guangjie Li, and Hui Liu. 2024. An empirical study on the potential of llms in automated software refactoring.

[39] Fang Liu, Yang Liu, Lin Shi, Zhen Yang, Li Zhang, Xiaoli Lian, Zhongqi Li, and Yuchi Ma. 2026. Beyond Functional Correctness: Exploring Hallucinations in LLM-Generated Code. arXiv:2404.00971 [cs.SE] https://arxiv.org/abs/2404.00971

[40] Yuhe Liu, Changhua Pei, Longlong Xu, Bohan Chen, Mingze Sun, Zhirui Zhang, Yongqian Sun, Shenglin Zhang, Kun Wang, Haiming Zhang, et al. 2025. OpsEval: A Comprehensive Benchmark Suite for Evaluating Large Language Models' Capability in IT Operations Domain. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*. ACM, 503–513.

[41] Lars Malmqvist. 2025. Sycophancy in large language models: Causes and mitigations. In *Intelligent Computing-Proceedings of the Computing Conference*. Springer, Springer, 61–74.

[42] João José Maranhão Junior, Filipe F Correia, and Eduardo Martins Guerra. 2024. Can ChatGPT Suggest Patterns? An Exploratory Study About Answers Given by AI-Assisted Tools to Design Problems. In *International Conference on Agile Software Development*. Springer Nature Switzerland Cham, Springer, 130–138.

[43] Boris Martinović and Robert Rozić. 2025. Perceived Impact of AI-Based Tooling on Software Development Code Quality. *SN Computer Science* 6, 1 (2025), 63.

[44] Ggaliwango Marvin, Nakayiza Hellen, Daudi Jjingo, and Joyce Nakatumba-Nabende. 2023. Prompt engineering in large language models. In *International conference on data intelligence and cognitive informatics*. Springer, Springer, 387–402.

[45] Tula Masterman, Sandi Besen, Mason Sawtell, and Alex Chao. 2024. The landscape of emerging ai agent architectures for reasoning, planning, and tool calling: A survey.

[46] Jeremy McHugh, Kristina Šekrst, and Jon Cefalu. 2025. Prompt Injection 2.0: Hybrid AI Threats. arXiv:2507.13169 [cs.CR] https://arxiv.org/abs/2507.13169

[47] Courtney Miller, Mahmoud Jahanshahi, Audris Mockus, Bogdan Vasilescu, and Christian Kästner. 2024. Understanding the response to open-source dependency abandonment in the npm ecosystem. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, ACM, 38–50.

[48] Iman Mirzadeh, Keivan Alizadeh, Hooman Shahrokhi, Oncel Tuzel, Samy Bengio, and Mehrdad Farajtabar. 2024. Gsm-symbolic: Understanding the limitations of mathematical reasoning in large language models. *arXiv preprint arXiv:2410.05229* (2024).

[49] Amr Mohamed, Maram Assi, and Mariam Guizani. 2025. The Impact of LLM-Assistants on Software Developer Productivity: A Systematic Literature Review. *arXiv preprint arXiv:2507.03156* (2025).

[50] Ziyi Ni, Huacan Wang, Shuo Zhang, Shuo Lu, Ziyang He, Wang You, Zhenheng Tang, Yuntao Du, Bill Sun, Hongzhang Liu, et al. 2025. GitTaskBench: A Benchmark for Code Agents Solving Real-World Tasks Through Code Repository Leveraging. *arXiv preprint arXiv:2508.18993* (2025).

[51] Marten Oltrogge, Erik Derr, Christian Stransky, Yasemin Acar, Sascha Fahl, Christian Rossow, Giancarlo Pellegrino, Sven Bugiel, and Michael Backes. 2018. The Rise of the Citizen Developer: Assessing the Security Impact of Online App Generators. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 634–647. doi:10.1109/SP.2018.00005

[52] 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*. ACM, 2464–2465.

[53] Sebastiano Panichella. 2024. Vulnerabilities introduced by llms through code suggestions. In *Large language models in cybersecurity: threats, exposure and mitigation*. Springer Nature Switzerland Cham, 87–97.

[54] Dorin Pomian, Abhiram Bellur, Malinda Dilhara, Zarina Kurbatova, Egor Bogomolov, Timofey Bryksin, and Danny Dig. 2024. Next-Generation Refactoring: Combining LLM Insights and IDE Capabilities for Extract Method. In *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 275–287. doi:10.1109/ICSME58944.2024.00034

[55] Karlis Rokis and Marite Kirikova. 2022. Challenges of Low-Code/No-Code Software Development: A Literature Review. In *Perspectives in Business Informatics Research*, Ērika Nazaruka, Kurt Sandkuhl, and Ulf Seigerroth (Eds.). Springer International Publishing, Cham, 3–17.

[56] Daniel Russo. 2024. Navigating the complexity of generative ai adoption in software engineering. *ACM Transactions on Software Engineering and Methodology* 33, 5 (2024), 1–50.

[57] Irfan Samsyudin. 2025. Vibe Coding and AI-Led Conversational Programming: Emerging Trends in Software Development. *Available at SSRN 5469367* (2025).

[58] Thomas Schuster, Marian Lambert, Nico Döring, and Julius Trögele. 2025. Needle-in-the-Haystack Testing LLMs with a Complex Reasoning Task. *Communications in Computer and Information Science* 2581 CCIS (2025), 254 – 266. doi:10.1007/978-3-031-96196-0_19 Cited by: 0.

[59] Idan Shenfeld, Jyothish Pari, and Pulkit Agrawal. 2025. RL's Razor: Why Online Reinforcement Learning Forgets Less. *arXiv preprint arXiv:2509.04259* (2025).

[60] Ken Shi and Gerald Penn. 2025. Semantic masking in a needle-in-a-haystack test for evaluating large language model long-text capabilities. In *Proceedings of the First Workshop on Writing Aids at the Crossroads of AI, Cognitive Science and NLP (WRAICOGS 2025)*. International Committee on Computational Linguistics, 16–23.

[61] Rui Song, Yingji Li, Lida Shi, Fausto Giunchiglia, and Hao Xu. 2024. Shortcut learning in in-context learning: A survey. *arXiv preprint arXiv:2411.02018* (2024).

[62] Jacob Mitchell Springer, Sachin Goyal, Kaiyue Wen, Tanishq Kumar, Xiang Yue, Sadhika Malladi, Graham Neubig, and Aditi Raghunathan. 2025. Overtrained Language Models Are Harder to Fine-Tune. arXiv:2503.19206 [cs.CL] https://arxiv.org/abs/2503.19206

[63] Ramya Srinivasan. 2025. *Misinformation and Disinformation in Generative AI—A Survey*. Lecture Notes in Computer Science, Vol. 15873. Springer Nature Singapore, Singapore, 290–307. doi:10.1007/978-981-96-8183-9_23

[64] Tao Sun, Jian Xu, Yuanpeng Li, Zhao Yan, Ge Zhang, Lintao Xie, Lu Geng, Zheng Wang, Yueyan Chen, Qin Lin, et al. 2025. Bitsai-cr: Automated code review via llm in practice. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*. ACM, 274–285.

[65] Rebeka Tóth, Tamas Bisztray, and László Erdődi. 2024. LLMs in Web Development: Evaluating LLM-Generated PHP Code Unveiling Vulnerabilities and Limitations. In *Computer Safety, Reliability, and Security. SAFECOMP 2024 Workshops*, Andrea Ceccarelli, Mario Trapp, Andrea Bondavalli, Erwin Schoitsch, Barbara Gallina, and Friedemann Bitsch (Eds.). Springer Nature Switzerland, Cham, 425–437.

[66] Jian Wang, Zhen Li, Jixiang Qu, Deqing Zou, Shouhuai Xu, Ziteng Xu, Zhenwei Wang, and Hai Jin. 2025. MalPacDetector: An LLM-based Malicious NPM Package Detector. *IEEE Transactions on Information Forensics and Security* (2025).

[67] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.

[68] Jiaheng Wei, Yuanshun Yao, Jean-Francois Ton, Hongyi Guo, Andrew Estornell, and Yang Liu. 2024. Measuring and Reducing LLM Hallucination without Gold-Standard Answers. arXiv:2402.10412 [cs.CL] https://arxiv.org/abs/2402.10412

[69] Orion Weller, Michael Boratko, Iftekhar Naim, and Jinhyuk Lee. 2025. On the Theoretical Limitations of Embedding-Based Retrieval. *arXiv preprint arXiv:2508.21038* (2025).

[70] Michael Wooldridge. 1997. Agent-based software engineering. *IEE Proceedings-software* 144, 1 (1997), 26–37.

[71] Elizabeth Wyss, Dominic Tassio, Lorenzo De Carli, and Drew Davidson. 2025. Evaluating LLM-Based Detection of Malicious Package Updates in npm. https://ldklab.github.io/assets/papers/raid25-llmdetection.pdf

[72] Lin Yang, Chen Yang, Shutao Gao, Weijing Wang, Bo Wang, Qihao Zhu, Xiao Chu, Jianyi Zhou, Guangtai Liang, Qianxiang Wang, and Junjie Chen. 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* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 1607–1619. doi:10.1145/3691620.3695529

[73] Wenqian Ye, Guangtao Zheng, Yunsheng Ma, Xu Cao, Bolin Lai, James M Rehg, and Aidong Zhang. 2024. Mm-spubench: Towards better understanding of spurious biases in multimodal llms.

[74] Nusrat Zahan, Philipp Burckhardt, Mikola Lysenko, Feross Aboukhadijeh, and Laurie Williams. 2024. MalwareBench: Malware samples are not enough. In 2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR).

[75] Nusrat Zahan, Philipp Burckhardt, Mikola Lysenko, Feross Aboukhadijeh, and Laurie Williams. 2025. *Leveraging Large Language Models to Detect npm Malicious Packages*. IEEE Press, 2625–2637. https://doi.org/10.1109/ICSE55347.2025.00146

[76] Ruediger Zarnekow and Walter Brenner. 2005. Distribution of cost over the application lifecycle-a multi-case study. *ECIS 2005 proceedings* (2005), 26.

[77] Zhiqing Zhong, Shilin He, Haoxuan Wang, Boxi Yu, Haowen Yang, and Pinjia He. 2025. An Empirical Study on Package-Level Deprecation in Python Ecosystem . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, USA, 66–77. doi:10.1109/ICSE55347.2025.00046

[78] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small world with high risks: A study of security threats in the npm ecosystem. In *28th USENIX Security symposium (USENIX security 19)*. 995–1010.