The Role of Code Proficiency in the Era of Generative Al

Gregorio Robles Universidad Rey Juan Carlos Madrid, Spain grex@gsyc.urjc.es

Jesus M. Gonzalez-Barahona Universidad Rey Juan Carlos Madrid, Spain jgb@gsyc.urjc.es

ABSTRACT

At the current pace of technological advancements, Generative AI models, including both Large Language Models and Large Multimodal Models, are becoming integral to the developer workspace. However, challenges emerge due to the 'black box' nature of many of these models, where the processes behind their outputs are not transparent. This position paper advocates for a 'white box' approach to these generative models, emphasizing the necessity of transparency and understanding in AI-generated code to match the proficiency levels of human developers and better enable software maintenance and evolution. We outline a research agenda aimed at investigating the alignment between AI-generated code and developer skills, highlighting the importance of responsibility, security, legal compliance, creativity, and social value in software development. The proposed research questions explore the potential of white-box methodologies to ensure that software remains an inspectable, adaptable, and trustworthy asset in the face of rapid AI integration, setting a course for research that could shape the role of code proficiency into 2030 and beyond.

ACM Reference Format:

1 INTRODUCTION

Artificial Intelligence, particularly large language models (LLMs), is changing how we think about building and maintaining software [16]. Some voices predict that in a not-too-distant future it will no longer be necessary to know the ins and outs of a programming language to be able to perform software engineering tasks [10]. Some others are concerned about the lack of control that this change could have for humans in charge of ensuring software systems work as intended [9]. From a historical perspective, some

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SE 2030, November 2024, Porto de Galinhas (Brazil) © 2024 Association for Computing Machinery. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...\$15.00

https://doi.org/10.1145/nnnnnn.nnnnnn

Christoph Treude

Singapore Management University Singapore, Singapore, Singapore ctreude@smu.edu.sg

Raula Gaikovina Kula Nara Institute of Science and Technology Nara, Japan raula-k@is.naist.jp

of these issues are reminiscent of the discussions of several decades ago, when compilers allowed us to abstract from assembly code and use higher-level languages. At first, many of the (assembler) programmers of that time did not trust the compilers and checked that the translation was done correctly. As work with high-level languages became more popular, this concern decreased, but the problem of to what extent compiler-produced code reflects the intentions of the programmer remained for a long time [1]. Nowadays, this is overcome, confidence in code produced by compilers is commonplace, and only in very specific cases is it necessary for humans to work at the assembler level. However, the situation now may be different, because we do not understand how generative Als produce code with the same detail that we understand how compilers produce code. In addition, the gaps between prompts and the code they produce, and the gap between source code in a high-level programming language and the code produced when compiling it, are very different, much higher in the first case.

As famous physicist Niels Bohr presumably said, predicting is complicated, especially the future. It may be the case that software engineers of the future will not need programming knowledge to perform their tasks. We call this case the "black-box case". But it could also be that they still need to read and understand the source code in a programming language. We call this case the "white box case". And, of course, the future could be (and likely will be) complicated, with a range of situations between these cases, depending on the scenario [19].

In the "pure" black-box case, software engineers will work at a very high level of abstraction, likely using natural language to transmit specifications to an AI agent, which will in turn produce computer programs ready to execute. They could be directly binary code or produced with the intermediation of some source code in traditional programming languages, but not visible to the engineer. Since specifying complex behaviors in natural languages is prone to misunderstandings, it is likely that the desired solution will not be achieved in a single iteration, so the engineer should check the output and reformulate or expand the prompt until the desired software product is achieved. Therefore, the complete process could be similar to a golf game, with successive shots getting us closer and closer to the final goal. Several methodologies have been proposed along these lines, such as, for example, the STL (specify-test loop) approach [4].

In the "pure" white-box case, software engineers will likely use AI agents to write source code, but they will be more like companions in a pair programming team than anything else [17]. The engineer

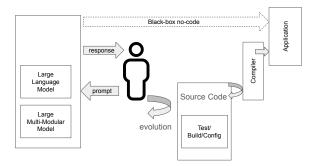


Figure 1: Overview of black-box vs. white-box when using AI agents. The upper part of it shows the black-box case: the software is directly produced by the LLM model, with no human intervention (except for prompting it, and interacting with the resulting software). The lower part of if shows how the white-box case includes humans in the loop, letting them inspect the produced source code.

will still read source code in a high-level programming language, touch it when needed, and in general use agents to help write and explain code. Engineers will have access to all implementation details and will need to comprehend them. This case is in the end quite similar to the current situation, but with much more powerful tools helping both to write and to understand source code.

We can explore in some more detail the differences between the white-box and black-box cases. Figure 1 shows a conceptual diagram that highlights the key distinctions between them with respect to how they relate to human software engineers. In a black-box scenario, a human uses a prompt to generate some software. Perhaps there is an intermediary step, where the source code produced by the LLM is compiled into object code. But the central idea is to avoid any human interaction since the prompt is provided up to the moment the software is produced. In contrast, the white-box approach allows a human to inspect the details of the outputs generated by the AI agent. This model positions a human between the LLM and the source code, facilitating multiple iterations of prompting to refine and evolve the desired code output.

Faced with these two extreme scenarios, we consider that usually the white-box approach will be preferred, even when for some scenarios black-box approaches can also be useful and appropriate.

2 EVOLUTION AND TRUST

One of our key reasons for favoring a white-box approach lies in the very nature of software development and maintenance. To remain useful, software needs to evolve [11]. It has to be constantly modified because of changes in expectations and requirements by the humans using it and because of changes in the environment (changes in the software system on which the program works, in other components with which it interacts, in the hardware in which it runs, etc.). In a successful software project, at least 80% of the effort goes to its evolution and maintenance [2]. For example, studies show that old software projects, more than 30 years old, with a significant size (in the range of several million lines of code)

have a code base with a large proportion of lines written recently (between 30% and 50% are less than 5 years old) [18].

The software that we run today requires constant updating. Contrary to what many might intuitively think, software is not usually the result of being created from scratch, and then have just a few specific changes. On the contrary, software is usually the result of many, very profound changes, many of them performed recently. In this sense, software differs from texts written in natural language, which are only occasionally modified to keep them current. In software, although the main aim of a program may remain the same since its beginning, changes are radical, both in what has been added and in the corrections.

In Table 1, we present concrete evidence illustrating the substantial maintenance effort needed to maintain and evolve the software. The data from a set of Free/Open Source Software (FOSS) projects show that i) although the current size of the codebase exceeds its original dimensions, the volume of code modifications applied (both additions and deletions) surpasses even the final size of the product itself, and ii) that the amount of recent code (i.e., lines included later than 2018) is a significant part of the current body of source code. These observations underscore the fact that code is a dynamic entity, constantly shaped by external and internal bugs, the introduction of new features, and the evolving requirements of users.

And we know that maintaining software requires knowledge and skills that are different from creating software [8]. They are difficult to exercise with a black-box approach because in many cases even specifying what exactly should be changed is quite a challenge without access to the source code. In those cases, being able to see inside the box, comprehending implementation details, is almost certainly necessary.

Another key reason for the white box case is that engineers will not blindly trust the code produced by AI agents, or that they will find too difficult or time-consuming to interact with until they are sure that the produced code actually performs as they intend. Current generative AI agents are prone to hallucinations [14], and may produce code with bugs [20]. In addition, specifying detailed behaviors in natural language is difficult due to its inherent lack of precision, the possibility of different interpretations, and the challenges of expressing complex actions. We also face the problem that, up to now, models are not explainable. That is, we do not completely understand how they are going to react to specific prompts or how small changes in prompts affect the results produced by a generative model. Furthermore, the issue of reproducibility and non-determinism in AI-generated code adds another layer of complexity, as the same prompt might not always result in the same output, making it challenging to ensure consistent and reliable software behavior.

If we cannot trust the code produced by AI agents at least at the same level as we trust the code produced by humans, we are going to want software engineers to check the code they produce and eventually fix it. This will be fundamental in fields such as safety-critical systems which are subject to specific certification requirements, which usually require that the code has been inspected in detail. But even in less demanding fields, being reasonably sure that the software is going to behave as expected may be challenging without at least the possibility of inspecting the code.

Table 1: Four FOSS projects showing to which extent their source code is recent. "LoC 2024" offers the lines of code in their most recent version. "LoC 2024 introduced after 2018" is the number of LoC in 2024 that have been introduced in 2018 or later (i.e., has been added in the last six years). "LoC all history" gives the number of LoC added in all the history of the project. "Removed LoC all history" is the LoC that have been removed. LoC stands for Lines of Code (considering comments but not blank lines).

	LoC 2024	LoC 2024 introduced after 2018	LoC all history	Removed LoC all history
gimp	1,050K	410K	2,714K	1,664K
gtk	898K	639K	1,808K	910K
http	451K	101K	893K	442K
wine	4.8M	2.5M	7.6M	2.8M

3 ADVANTAGES OF THE WHITE-BOX PARADIGM

We argue that adopting a white-box paradigm for software maintenance, evolution, and even initial creation offers several advantages.

- a) **Responsibility.** In some sense, code acts as law [12]. Software acts, to some extent, as contracts do in the legal world, and nobody would like to sign an opaque contract, although very often only lawyers can understand the legal jargon. It is not difficult to argue for the importance of software in the daily life of modern society, and while errors in software are inevitable, it is not the same if they are the result of human errors, or a consequence of being no humans verifying that software behaves as intended. A white-box approach, where software can be (and is) inspected and understood by humans, seems like a reasonable requirement for this responsibility.
- b) **Security.** The possibility of working at the source code level allows security audits to be carried out on the created software systems. The argument in this regard is similar to what open source proponents advocate about the advantages of being able to see the source code: security by inspection versus security by obscurity. To be able to audit the security of software, the code produced by an LLM should be of high level and understandable by software developers.
- c) **Legal compliance and avoidance of bias** Similarly, if the code cannot be seen, it cannot be certified that the software system does what it should do or, on the contrary, does not do what it should not do. For the first thing, access to the source code is essential, and that it can be verified, by humans with the help of tools, that it is correct and that it meets certain parameters of quality and stability, among others. For the second, it is also necessary to be able to access the source code and verify that the software does not contain bias towards people or groups.
- d) Creativity. Seeing source code is a trigger for creativity. Seeing the current state of the source code can be a source of new inspiration, new possibilities, and innovation [5]. Creativity is clearly limited in a black-box environment. In this sense, software development, particularly free software, has been classified as a stigmatic process in which the source code plays the role of signaling. In other words, just as changes in the environment cause insects such as ants to behave differently, it has been argued that changes in source code have a pulling power for new changes. That is why, even without embracing free software models, there have been movements by large companies to show the source code of their software, so that people external to these companies can contribute.

e) **Social value.** In recent years, we have seen a revolution in software development toward social development mechanisms [3]. Thus, the software gains value through the exchange of opinions, collective supervision, and the possibility of remixing existing source code (forking). Social software development requires, for optimal development, a white-box environment. This does not mean that social development cannot be carried out to improve prompt engineering, but this would still be one of the many fields where software development benefits from it.

Our position, therefore, comes to say that software development, and especially its maintenance and evolution, is in constant exchange that creativity is maximized, as well as social value. It is this exchange that offers the necessary trust - when many other developers have seen it - that allows certain degrees of responsibility to be offered.

4 PROFICIENCY IS KEY

Of all the implications of the previous discussion, we will focus on one that we consider of utmost importance: the proficiency in understanding source code. In the white-box scenario, source code generated by LLMs must be not only accessible but also understandable by developers. Of course, developers may be assisted by tools, some of them being AI agents themselves, which help them to check for functional or non-functional properties of the produced code. But they should also be able to understand and reason about the code produced.

In other words, in addition to giving precise instructions about what we want an LLM to generate, questions related to the skills that the developer or development team possesses must also be included as input to the model. This goes beyond the programming language in which the LLM must generate code, as well as other constraints such as the style guide and the legibility of the source code. In addition, it is fundamental that the code produced by the AI agent is adjusted to the level of knowledge of the software engineers who will interact with it.

For natural languages (English, German, Spanish, etc.), there are frameworks to determine the level of proficiency of a language. For example, the CEFR, or Common European Framework of Reference for Languages, is an international reference framework that describes levels of proficiency in a foreign language in Europe [7], although there are other frameworks with a similar philosophy in other countries [13, 15]. It is divided into six levels: A1, A2, B1, B2, C1, and C2, from beginner to advanced. The CEFR is a useful

tool to assess student progress, compare levels between different languages, and facilitate student and worker mobility.

If we ask an LLM for a text or description in a foreign language, we would need it to be produced in accordance to our skills in that language if we want to understand it and maybe modify it. English Wikipedia, for example, has a Simple English Wikipedia project [6], with requirements such as "articles should use only the 1,000 most common and basic words in English. They should also use only simple grammar and shorter sentences."

We consider that it will be necessary as well to specify how we want AI agents to produce their code: which constructs they will use, which level of complexity, and, in general, which elements of the programming language. In this way, we could adapt the output of the LLM to the skills and capabilities of the engineers that should interact with it, in a way that favors its comprehension and makes it easier for them to understand and, if needed, change it. Therefore, it will be important to create proficiency level assessment frameworks for programming languages, just as has been done for natural languages.

This would allow developers to understand, analyze, adapt and modify the code generated by the LLMs, maximizing the advantages of the white-box point of view, as we have argued in the previous paragraph. So, if a developer has a B2 (intermediate) level of Python, they should be able to ask the LLM for a piece of code that performs an algorithm or corrects a bug at "Python level B2". LLMs would be trained with these requirements in mind, for example, automatically labeling their training set with the corresponding "proficiency level". Of course, the LLM can be customized not only in terms of the level of mastery, but also in terms of the style guide, the size and details of comments (and the natural language used for them), for example.

For software projects produced from scratch, the proficiency level of the source code could be defined in advance, given the expected skills of the engineers working on that project. For already existing software projects, an analysis of the level of their source code could be produced, so that further output by AI agents matches that level, which engineers are accustomed to.

5 RESEARCH AGENDA

We outline a research agenda below, presenting specific questions to guide further exploration of white-box vs. black-box approaches in the era of generative AI and their implications for code proficiency.

Code as a reflection of developer skills. The inquiry into how the quality of code produced by a program reflects the developer's expertise delves into understanding the effectiveness of automated programming tools in relation to the user's skill level. Hence, we ask the first two research questions:

Skills

- How well does the quality of code produced by a program mirror the developer's expertise?
- How well does the number of iterations required to produce high-quality code reflect developer skill level?

Leveraging Competency with AI. Our next research questions aim to investigate how AI-generated code, when aligned with developer proficiency levels within a white-box framework, impacts various dimensions of software development, including responsibility, security, legal compliance, and the broader socio-technical ecosystem.

Responsibility

- How can a white-box approach to AI-generated code help delineate responsibility for code outcomes when multiple proficiency levels are involved?
- In what ways does aligning AI-generated code with developer proficiency levels affect the attribution of responsibility for the code's quality and its outcomes?

Security

 How does tailoring AI-generated code to match developer proficiency levels within a white-box framework impact the code's resilience against security vulnerabilities?

Legal

 How can ensuring that AI-generated code is adaptable to various levels of developer proficiency within a whitebox approach mitigate legal risks and reduce biases in software?

Creativity

- Is there a tendency for high-quality code to exhibit greater creativity?
- Can AI-generated code lead to a reduction in diverse problem-solving approaches?

Social

- Do developers have the necessary skills to review automatically generated code effectively?
- Does code that reflects the creator's proficiency improve developers' trust and perception, even without human verification?

6 CONCLUSION

Developers integrating AI into their software development environments is not just a trend but an inevitability, many having already embraced this shift. This paper has outlined various concerns associated with AI in software development and has argued for the critical importance of adopting a white-box approach. By ensuring that source code serves as a transparent measure of proficiency and understanding, we underscore its vital role in the future of software development as we move towards 2030 and beyond.

REFERENCES

- James M Boyle, R Daniel Resler, and Victor L Winter. 1999. Do you trust your compiler? Computer 32, 5 (1999), 65–73.
- [2] Shi Kuo Chang. 2001. Handbook of software engineering and knowledge engineering. Vol. 1. World Scientific.
- [3] Cleidson De Souza, Jon Froehlich, and Paul Dourish. 2005. Seeking the source: software source code as a social and technical artifact. In Proceedings of the 2005 ACM International Conference on Supporting Group Work. 197–206.
- [4] Jesus M. Gonzalez-Barahona. 2024. Software development in the age of LLMs and XR. In Proceedings of the First IDE Workshop (IDE âĂŽ24). ACM.
- [5] Wouter Groeneveld, Laurens Luyten, Joost Vennekens, and Kris Aerts. 2021. Exploring the role of creativity in software engineering. In 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Society (ICSE-SEIS). IEEE. 1-9.
- [6] William Hwang, Hannaneh Hajishirzi, Mari Ostendorf, and Wei Wu. 2015. Aligning sentences from standard wikipedia to simple wikipedia. In Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. 211–217.
- [7] Neil Jones and Nick Saville. 2009. European language policy: Assessment, learning, and the CEFR. Annual Review of Applied Linguistics 29 (2009), 51–63.
- [8] Magne Jørgensen and Dag IK Sjøberg. 2002. Impact of experience on maintenance skills. Journal of Software Maintenance and Evolution: Research and Practice 14, 2 (2002), 123–146.
- [9] Vassilka D KIROVA, CS Ku, JR Laracy, and TJ Marlowe. 2023. The Ethics of Artificial Intelligence in the Era of Generative AI. Journal of Systemics, Cybernetics and Informatics 21, 4 (2023), 42–50.
- [10] Mohammad Amin Kuhail, Sujith Samuel Mathew, Ashraf Khalil, Jose Berengueres, and Syed Jawad Hussain Shah. 2024. âĂIJWill I Be Replaced?âĂİ Assessing

- ChatGPT's Effect on Software Development and Programmer Perceptions of AI Tools. Science of Computer Programming (2024), 103111.
- [11] Manny M Lehman. 1996. Laws of software evolution revisited. In European workshop on software process technology. Springer, 108–124.
- [12] Lawrence Lessig. 2000. Code is law. Harvard magazine 1 (2000), 2000.
- [13] Judith E Liskin-Gasparro. 1984. The ACTFL Proficiency Guidelines: A Historical Perspective. (1984).
- [14] Fang Liu, Yang Liu, Lin Shi, Houkun Huang, Ruifeng Wang, Zhen Yang, and Li Zhang. 2024. Exploring and Evaluating Hallucinations in LLM-Powered Code Generation. arXiv preprint arXiv:2404.00971 (2024).
- [15] Hitoshi Nishizawa, Daniel R Isbell, and Yuichi Suzuki. 2022. Review of the Japanese-language proficiency test. Language Testing 39, 3 (2022), 494–503.
- [16] Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. 2023. The impact of ai on developer productivity: Evidence from github copilot. arXiv preprint arXiv:2302.06590 (2023).
- [17] Peter Robe, Sandeep K Kuttal, Jake AuBuchon, and Jacob Hart. 2022. Pair programming conversations with agents vs. developers: challenges and opportunities for SE community. In Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 319–331.
- [18] Gregorio Robles, Jesus M Gonzalez-Barahona, and Israel Herraiz. 2005. An empirical approach to software archaeology. In Proc. of 21st Int. Conf. on Software Maintenance (ICSM 2005), Budapest, Hungary. 47–50.
- [19] Jaakko Sauvola, Sasu Tarkoma, Mika Klemettinen, Jukka Riekki, and David Doermann. 2024. Future of software development with generative AI. Automated Software Engineering 31, 1 (2024), 26.
- [20] Florian Tambon, Arghavan Moradi Dakhel, Amin Nikanjam, Foutse Khomh, Michel C Desmarais, and Giuliano Antoniol. 2024. Bugs in large language models generated code. arXiv preprint arXiv:2403.08937 (2024).