Large Language Models in Software Engineering Processes

Preprint · September 2024

DOI: 10.13140/RG.2.2.29409.11361

1 author:

Mohamed Khalil Brik
American University in Cairo
1 PUBLICATION **0** CITATIONS
SEE PROFILE

Large Language Models in Software Engineering Processes

Mohamed Khalil Brik
The American University in Cairo

September 2024

Abstract

In this paper, we will discuss how LLMs could be used in the process of software engineering. We will go through the main cycles such as design, code generation, testing, debugging, and team collaboration. We will discuss the importance of each cycle, the challenges faced by engineers, and the contributions that could be made by LLMs. Later we will discuss the issues when using Large Language Models in software engineering. Then we will introduce future research directions for further improvements of LLMs in creating software.

Keywords: large language models, software engineering, requirements, architecture, GPT, general AI, explainable AI.

I) Introduction

Large Language Models (LLM) are a subset of foundational models trained on a massive amount of data primarily accessed from the internet. LLM models can understand natural language text and other types of content such as images, videos, and more. Relying on AI technologies like Deep Learning, LLMs can understand, summarize, and generate new content. The widespread of LLMs after the launch of the GPT 3 model by OpenAI allowed people to use this technology to increase their productivity in multiple domains such as education, healthcare, and software engineering. LLMs' capabilities go beyond interacting with human language to analyzing and producing computer programs. This feature allowed many software engineers to use powerful LLMs such as the GPT model or the Google Bard as their assistant tool in developing software. LLM's use cases go beyond writing code to assisting in various software engineering cycles such as Requirements Engineering, System Design, Code Generation, Documentation, and more. In his research, Wei introduced a tailored LLM to go through all software engineering tasks after requirements have been prompted to the model [8]. Such results are revolutionizing our perspectives towards the entire software industry. Throughout this paper, we will look at the different contributions LLMs can make to the various software development phases. We will also study the limitations that face modern LLMs. These limitations created future directions for research in this field to investigate.

II) LLMs in Software Development Phases

1) Coding and Implementation

Generating code is a fundamental step in the software development lifecycle. It is also one of the most time-consuming steps. A report by TechRepublic shows that developers spend 25% to 35% of their working time writing computer codes [7]. Nevertheless, most of this time is spent writing code for repetitive tasks that developers replicate through multiple projects. For instance, front-end developers usually write the same code to design UI components such as forms, buttons, and drop-down menus. In server-side programming, developers constantly write the same code to create APIs, database connections, and authentications with minor differences between projects. LLM-powered products such as GitHub Copilot are very efficient in handling repetitive tasks. Users of GitHub Copilot reported time savings of up to 55% when using it to generate repetitive snippets of code [5]. The current trend in software development is to integrate LLM products into the development ecosystem. Integrated Development Environments (IDEs) such as VS Code provide GitHub Copilot as an extension that developers can use to generate repetitive code from comments. Besides, Copilot offers code suggestions in real-time, which eliminates defects before being integrated into the codebase [5]. By automating repetitive tasks, developers can focus on more complex tasks that require higher levels of human intelligence and creativity.

2) Testing and Debugging

Code testing and debugging are crucial cycles in software development that are considered a nightmare for many developers. Translating the architecture of software into code is often prone to human errors due to miscommunications, confusion, or the inexperience of novice developers. The higher the complexity of the architecture, the more of these errors are to be found. As a result, testing ensures that software performs as expected and matches the behavior described in the architecture. Testing is also a part of Quality Assurance to ensure that software meets requirements, is reliable, and satisfies user expectations. LLMs can generate exhaustive test cases that cover a broad surface of software functionalities relying on system requirements. Unit tests generated by LLMs cover most corner cases of functions that could be missed by developers. LLMs could also simulate users' interactions with the interface based on use case scenarios defined by requirements. LLMs can also increase the inclusivity of software by simulating users with different needs, such as the visually impaired, which could be harder to test by engineers. Research shows that developers using LLM-produced test cases by Copilot saw an average of 50% decrease in manual testing efforts [5].

When a test case fails, developers start the debugging process to discover the root cause of the failure. Debugging involves multiple steps such as reproducing the bug with a different test case, identifying the problematic piece of code, and fixing the bug. LLMs can help in locating the code causing the defect, especially in very complex software products where manual search for errors becomes challenging. Research shows that ChatGPT-4 outperforms classical state-of-the-art baseline SmartFL (a classical ML model) by 45.9% on average in fault localization tasks [9]. This research proves that LLMs are replacing classical Machine Learning Models that require a harder training process and manual parameter tuning.

3) Documentation and Code Review

Documentation is an essential step in the software development cycle to ensure that software can be modified by future developers different from the original creation team. Engineers in big tech companies such as Google spend most of their time improving existing software products. This process requires access to well-structured documentation. Good documentation generally includes step-by-step instructions to explain how a specific functionality was engineered in clear language. A study by the Consortium for IT Software Quality (CISQ) proved that poor documentation leads to software defects that cost companies up to 59 billion dollars in the US alone [6]. LLMs can generate full documentation relying on comments written by developers and the structure of the software. LLMs can generate high-level documentation of software products that are language-independent and can be presented to non-technical stakeholders. For example, a tech company can train their own tailored LLM on the software they own. The LLM will have real-time access to the entire code base and existing documentation. As soon as new functionality is integrated, the LLM will generate new documentation for the code efficiently as it is fine-tuned to the specific software it was trained on. Company employees can interact with this LLM through a chatbot, which helps new developers learn about the software rather than asking senior developers. This approach not only saves time in all software cycles but also increases precision and efficiency when merging existing features with new ones. An empirical study by Dvivedi et al. concluded that "All Large Language Models (LLMs) exhibit either equal or superior performance compared to the original documentation." The LLMs used in the study include GPT-4, Bard, Llama2, and others [12].

Another important aspect of software development is code reviews performed during the development process. Before new features are deployed, the logic, structure, and code performance are evaluated against multiple metrics. This review ensures that the code meets all the industry standards of quality, functionality, and security. LLMs can improve the code review process by providing suggestions for improvement in real-time. Besides, relying on Explainable Artificial Intelligence (XAI), LLMs can provide developers with the reasons behind their suggestions, allowing engineers to learn about better practices and avoid future defects.

4) Software Design and Architecture

Software Design and Architecture are very critical steps in the cycle. In this stage, functional and non-functional requirements are translated into high-level architecture that the development team will reference later to produce the software code. Architects often struggle with a vague and unclear set of requirements, which can lead to design flaws and inconsistencies in the system. To solve this issue, architects communicate with requirements engineers back and forth until they resolve defects in the requirements specifications. In 2013, the healthcare.gov website went down two hours after its launch due to the millions of Americans who tried to register for health insurance through this website. The architecture of the website lacked integration between the different components of the software, which were designed by multiple contractors. This led to user information not being processed correctly in transit from different sub-systems, resulting in registration failures. The cost of fixing this system after the first failure reached 1.7 billion dollars [1].

LLMs are being used to generate full software architecture from requirements. They can also point out problems in the requirements by asking clarification questions to the requirements team before designing the full architecture. LLMs can aid architects by suggesting alternative designs, identifying trade-offs, and listing potential defects in the architecture. The LLM will produce the architecture as design diagrams using Text-to-Image models such as DALL·E. A study showed that the time needed to create high-level architecture decreased by up to 40% when using OpenAI's DALL·E [5]. In their research, Belzner et al. prompted ChatGPT to generate UML diagrams after specifying software requirements. The LLM provided them with class diagrams and pseudocode of the most important classes of the system [2]. Architects can also interact with LLMs to get new perspectives on architecture. They can ask the LLM questions such as "Are there risk areas within the architecture?", "How should the risk be addressed?", and "Are there any common anti-patterns in the architecture?". Using LLMs for creating a software architecture could be a very efficient brainstorming tool for engineers. LLMs can direct architects to consider unexplored scenarios or features that will make the architecture more optimized.

5) Team Collaboration

Many software defects could have been easily avoided if properly structured channels of communication were implemented. Modern practices in the software industry include relying on LLMs to improve team collaboration. We will further use the example of the LLM integrated into the software of a company previously described. This LLM can help software stakeholders with different backgrounds to get clarifications on questions they might have. For instance, a member of the business team can ask the LLM, "What is a recommended timeline for our development team to create a chatbot for our mobile application?" Considering that the LLM is well immersed in the work of all different teams, it will be able to consider all ongoing projects the development team is working on and suggest a realistic timeline to develop the chatbot feature. As a result, in further meetings, the business team will have realistic expectations from the development team regarding the creation of this chatbot.

Furthermore, LLMs can improve team collaboration within the same team. For example, a junior software developer, Mark, who recently joined the company can ask the LLM about the architecture of the payment API used in the software. This approach is more efficient and time-saving for Mark. Instead of asking a senior member of the team who might not give him the full picture, Mark will instead start a conversation with the LLM to answer any further questions he might have about the exact code executing the described functionality.

Another aspect of team collaboration is using version control systems (VCS), which enable many developers to work on the same codebase concurrently. An LLM such as GitHub Copilot can help developers adhere to best practices in version control. For example, an LLM can generate meaningful commit messages that best represent the current state of the codebase. Another example is assisting in conflict resolutions. When two commits are in conflict, a merge conflict arises, which is usually processed by a senior developer. An LLM can explain the difference between the two conflicting commits, analyze their impact, and even suggest how they could be integrated.

III) Challenges and Limitations

While integrating LLMs in the different software cycles has a large positive impact on software development, some challenges and limitations are associated with using this technology.

First, LLMs could produce inaccurate results, generating syntactically correct code that is not functional and does not perform the required task. This could be due to a lack of full understanding of the project. Developers usually prompt the LLM with specific parts of the codebase, which can lead to generating code that cannot be integrated with the entire codebase.

Another key risk when using LLMs is over-reliance on the output. An LLM can produce a class that seems correct, but it may cause significant damage if integrated into the system. A novice programmer might blindly trust the generated code and include it in the codebase without further verification. This risky behavior could lead to major defects, resulting in downtime or significant security threats.

One of the interesting behaviors of LLMs is misalignment. LLMs are trained on a large amount of data collected from the internet; however, the quality of this data is not evaluated by an experienced engineer. As a result, the training distribution could contain a percentage of code that damages systems. Additionally, the code provided to the LLM by the user could have several defects overlooked by the programmer. LLMs do not have the conscious ability to differentiate between good and bad code. Consequently, they might deliberately produce incorrect code, which is referred to as alignment failure [3]. It is essential to note that misalignment occurs when the model can produce correct code but "chooses" not to. Research by Fu et al. fine-tuned a GPT LLM, which they called Codex, on publicly available code from GitHub [3]. The model's accuracy was around 0.3 when the prompt included only correct examples. However, this accuracy decreased to 0.22 when the Codex was prompted with examples containing subtle bugs in context [3].

With the production of incorrect code by LLMs, security risks cannot be avoided. Fu et al. analyzed the security risks associated with Python and JavaScript codes generated by GitHub Copilot. From 277 code snippets written in Python by Copilot, 32.8% contained security risks; for the 175 JavaScript snippets, 24.6% also had risks. The risks identified include 38 Common Weakness Enumeration (CWE) weaknesses, which cover 10% of all CWEs (439 CWEs) [10].

Following the modern global issue of climate change, considering the environmental impact of using LLMs in software development is necessary. Training a Large Language Model, such as GPT, could have a significant carbon footprint. For instance, GPT-3 consumed around 1284 megawatt-hours of electricity, resulting in over 500 tons of CO2 emissions [13]. This amount of carbon emissions is equivalent to driving 50 cars to the moon and back.

IV) Future Directions and Opportunities

Further research contributions are needed to improve Large Language Models as a whole and as a software tool specifically. The datasets used to train the LLMs need to be enhanced. Larger datasets should cover different programming languages, various programming styles, and multiple coding principles. Besides, researchers should consider finetuning LLMs to specific programming languages. Training an LLM based on one language or engineering methodology will increase the accuracy and decrease coding errors. We could also integrate domain knowledge into LLMs. As a result, we will have specialized LLMs in web development, mobile apps, and machine learning. This could be achieved by prompt engineering to model the practices and the guidelines it needs to follow. This approach will limit the LLM from producing harmful code with large security threats.

We should also consider improving open-source LLMs compared to their closed counterparts. LLMs development should be a collective contribution from the entire research community worldwide. Open source LLMs could be finetuned by different experts in various domains. Creating multiple open-source LLMs could allow researchers to compare them for further improvement. It can also solve the privacy problem with closed-source LLMs. Users can access the source code and learn how their prompt is being used and whether it is being stored in a database or not. Companies and educational institutions should teach current and future software engineers the right way to use LLMs. The blind trust in LLMs is becoming a huge problem that needs to be addressed. Students should learn how to review any code produced by LLMs and how to integrate it into the full project. A complete ban on using LLMs in education will not benefit students' long term. Students should learn how to co-work with AI to increase their productivity and their learning curve at the same time.

V) Conclusion

In this paper we introduced how LLMs could be used in the different cycles of software engineering such as architecture, design, coding, and documentation. We referenced different case studies by researchers evaluating LLMs performance in creating software against multiple metrics such as accuracy, bias, speed... We also provided hypothetical use cases for LLMs in the context of software companies. These promising results did not prohibit us from studying the limitations and challenges in this rising technology. Problems such as over-reliance, inaccurate models, misalignment were introduced. In our last section of this paper, we discussed future research directions and opportunities to improve the overall performance of LLMs and software developers using them.

References

[1] ABC123. "The Failed Launch of Www.Healthcare.Gov." *Technology and Operations Management*, 18 Nov. 2016, https://d3.harvard.edu/platform-rctom/submission/the-failed-launch-of-www-healthcare-gov/.

[2] Belzner, Lenz, Gabor, Thomas, and Wirsing, Martin. "Large Language Model Assisted Software Engineering." 2023.

[3] Chen, M., et al. "Evaluating Large Language Models Trained on Code." *GitHub Copilot and OpenAI Codex*, 2021.

[4] "Comparing LLM Benchmarks for Software Development." *Symflower for IntelliJ IDEA - Smart Unit Test Generator for Java*, symflower.com/en/company/blog/2024/comparing-llm-benchmarks.

[5] Kalliamvakou, Eirini. “Research: Quantifying GitHub Copilot’s Impact on Developer Productivity and Happiness.” *The GitHub Blog*, 21 May 2024, github.blog/news-insights/research/research-quantifying-github-copilots-impact-on-developer-productivity-and-happiness.

[6] Krasner, Herb. “The Cost of Poor Software Quality in the US: A 2020 Report.” *CISQ*, 2020, www.it-cisq.org/the-cost-of-poor-software-quality-in-the-us-a-2020-report.

[7] Vigliarolo, Brandon. “Majority of Developers Spending Half, or Less, of Their Day Coding, Report Finds.” *TechRepublic*, 7 May 2019, www.techrepublic.com/article/majority-of-developers-spending-half-or-less-of-their-day-coding-report-finds.

[8] Wei, Bingyang. “Requirements Are All You Need: From Requirements to Code with LLMs.” *arXiv*, 2024, doi:10.48550/arXiv.2406.10101.

[9] Wu, Yonghao, et al. “Large Language Models in Fault Localisation.” *arXiv*, 2023, doi:10.48550/arXiv.2308.15276.

[10] Fu, Yujia, et al. “Security Weaknesses of Copilot Generated Code in GitHub.” *arXiv.org*, 4 Apr. 2024, arxiv.org/abs/2310.02059.

[11] Zhang, Y., et al. “Security Risks of AI-Assisted Coding: A Comprehensive Study of GitHub Copilot’s Vulnerabilities.” *GitHub*, 2022.

[12] Dvivedi, Shubhang Shekhar, et al. “A Comparative Analysis of Large Language Models for Code Documentation Generation.” *arXiv*, 2023, ar5iv.labs.arxiv.org/html/2312.10349v1.

[13] Patterson, David, et al. ”Carbon Emissions and Large Neural Network Training.” *arXiv*, 21 Apr. 2021, https://arxiv.org/abs/2104.10350.