Chances and Challenges of LLM-based Software Reengineering

Jochen Quante and Matthias Woehrle

Bosch Research
Renningen, Germany

Large Language Models (LLMs) have opened up unforeseen new possibilities. They deliver amazing results for complex text-based tasks for which no satisfactory automated solution was available before. This is even more astonishing as they just calculate the most probable subsequent token, given a sequence of tokens. This is also true for software development support: There are various software engineering task for which LLMs show potential [1, 5]. In this paper, we discuss the potential and current shortcomings of LLM-based approaches for selected Software Reengineering tasks with a focus on language translation. All experiments reported in the following were performed with GPT-4.

1 Language Translation

Language translation has been performed based on abstract syntax trees (AST) for decades. A typical use case for language translation are legacy applications from banking or insurance domains that often need to be translated from COBOL to Java and migrated to a new platform. This is necessary as the old hardware systems and compilers can no longer be maintained, and as it is increasingly hard to find people who have the required competencies to maintain that kind of code. The problem of these AST-based approaches is that the resulting code does not look and feel like real Java code, as it does not use the idioms of the target language, but rather stays close to the style of the original language. For example, when transforming from COBOL to Java, classes will be used. However, they will not be used in a natural way, but rather as containers for the respective code.

In contrast to that, LLMs have the capability to do a different way of translation, as they take more context into account. These models can leverage knowledge of similar code of the target language that they have seen during training. In the best case, they "recognize" an algorithm and translate it to the same algorithm in the target language. This works particularly well for often-used algorithms like sorting and searching. However, things look different for custom code. And from our experience, this heavily depends on the languages used: The more code in a language was contained in the training data, the better it works. As an example, the Python capabilities of current LLMs often excel other languages.

As a concrete example, let us discuss the results of some experiments on C to Rust translation using an LLM. Let us first look at the advantages of the LLM-based approach: The code from GPT-4 looked much better (more Rust-like) than code that was translated using c2rust[1], a classical AST-based tool for this task. This is because c2rust does a 1:1 translation. That means that all code is wrapped in unsafe blocks and then the original C implementation is used, e.g., the C functions as well as C strings. In contrast to that, the LLM translation mostly uses the adequate Rust-specific language features. It creates "safe" code whenever possible, which enables much stronger memory safety checks by the compiler. It can also produce code in different flavors, e.g., you can ask the LLM to generate the code in a more functional style. However, there are also some disadvantages. The LLM-based approach sometimes fails to use the right types, e.g., built-in types of Rust. Moreover, it has problems with correctly translating global variables from C, which basically do not exist in Rust. This means that in many cases, the proposed code is not even compilable. Thus, the LLM-based translation often requires significant manual clean-up work or additional interaction with the LLM to get to a correct translation.

Another issue for current models arises with longer functions or even whole code bases that shall be translated. Due to the limited context size of current LLMs, they can only deal with text of limited size. For example, the version of GPT-4 we used has a context size of 8,192 tokens, which often means for the translation tasks that only functions on the order of 100 lines of code can be translated at once. Furthermore, LLMs have a random component: you get different results for the same prompt each time, even with the lowest possible temperature. This means when translating pieces of code iteratively (to deal with limited context size), these fragments do not necessarily fit together. Again, this results in manual work, as identifiers may be renamed differently, different types may be used, etc. Therefore, it is hardly possible to use LLMs for translating entire code bases without additional measures[2]. You must at least provide detailed instructions on how to deal with certain constructs, like global variables, to get a more consistent and syntactically

[1] `https://c2rust.com/`

[2] See Outlook below for comments on current developments.

and semantically correct set of translated functions and data structures.

Overall, LLMs deliver very promising translation results for small code snippets or individual functions. However, current models with context limits are hardly usable for translation of larger code bases.

2 Other Reengineering Tasks

Besides language translation, LLMs can support in many other code-related tasks [1, 5]. When asking GPT to propose **software refactorings** for a given piece of code, it usually comes up with a very generic list of things that could be done: Choose better identifier names, split the function into smaller ones, etc. However, when asking it to perform these refactorings on a concrete piece of code, its capabilities are quite limited. While simple things such as the introduction of better identifier names often works quite well if the code is sufficiently self-explanatory, more sophisticated refactorings like extracting methods fail completely or deliver incomplete code. For performing such refactorings, the classical approach still seems to be the better choice.

LLMs can also be asked to explain code, so they can potentially help in **program comprehension**. This works perfectly fine for well-known algorithms, even if names of the functions and identifiers are obfuscated. For custom code, you often get an explanation that sounds reasonable, yet often the explanation is either wrong or not helpful, as it then explains the code line by line. Hence, this is not a big advancement compared to state-of-the-art code summarization approaches on that level. Nevertheless, we should mention that there were recently remarkable advances in answering sophisticated questions about a whole code base with long context models [3]. These results are very encouraging and contrast our current experiments on smaller models.

Finally, let use mention several further promising tasks for reengineering support. These include automatic program repair [2], performance improvement [1], as well as code search [5]. For a comprehensive overview of the potential of LLMs for software engineering tasks in general, we refer to Fan *et al.* [1].

3 Outlook

Predictions are hard, especially about the future. This is particularly true in the fast-moving field of LLMs. As such, current failing applications of LLMs in Software Reengineering tasks may only be a current snapshot as technology evolves. As an example, we want to highlight the rapid progress with respect to context size. While the context size of current models may limit the capabilities in program comprehension as discussed above, recent demonstration of LLMs that allow to feed a complete codebase into the LLMs have shown remarkable capabilities [3]. It will be exciting to see whether improvements in LLMs and systems including LLMs can remedy current limitations. Nevertheless, we need to consider how developers will interact with such a system depending on the success rate of queries and what we can learn from other engineering fields that have seen similar increasing levels of automation [4].

4 Conclusion

LLMs offer completely new possibilities for Software Reengineering. Notably, they take a different approach compared to classical techniques and thus provide different advantages, e.g., adapting to coding style. Moreover, LLM-based approaches outperform classical techniques on several benchmark tasks. However, there are some open problems with these approaches. For example, we mentioned scalability above. More substantially, there is in general a lack of any guarantee about the result quality. This lack of guarantees is however an inherent property of any AI approach, and thus we – as a software (re-)engineering community, probably have to identify additional (non-AI) measures to address it. For now, LLMs are a programmer's companion that can be used to generate proposals. The developers must be in the loop and check if the result is really what they wanted. This interaction between the developer and the companion and corresponding increasing levels of automation [4] is an important challenge and may require even deeper skills from the developer.

References

[1] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang. Large language models for software engineering: Survey and open problems. In *Proc. of 45th Int'l Conf. on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, pages 31–53, 2023.

[2] Z. Fan, X. Gao, M. Mirchev, A. Roychoudhury, and S. H. Tan. Automated repair of programs from large language models. In *Proc. of 45th Int'l Conf. on Software Engineering (ICSE)*, pages 1469–1481. IEEE, 2023.

[3] M. Reid, N. Savinov, et al. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530*, 2024.

[4] A. Sellen and E. Horvitz. The rise of the AI Co-Pilot: Lessons for design from aviation and beyond. *arXiv preprint arXiv:2311.14713*, 2023.

[5] G. Sridhara, R. H. G., and S. Mazumdar. ChatGPT: A study on its utility for ubiquitous software engineering tasks. *arXiv preprint arXiv:2305.16837*, 2023.