IDE Native, Foundation Model Based Agents for Software Refactoring

Abhiram Bellur
University of Colorado, Boulder
Boulder, CO, USA
abhiram.bellur@colorado.edu

Fraol Batole
Tulane University
New Orleans, LA, USA
fbatole@tulane.edu

***Abstract*—Foundation Models (FMs) have been trained on a massive amount of coding data and, at their best, are capable of looking at code like an expert software developer. This has led researchers to explore the possibility of FM-based agents for various software engineering tasks. These agents are capable of planning and executing complex tasks, such as bug repair, in the manner that an expert developer who is familiar with the codebase would. However, FMs often produce puzzling responses that are capable of resulting in buggy or vulnerable code. Inspired by recent work in agents for software engineering tasks, we discuss the idea of a FM-based refactoring agent, which is capable of scanning the entire codebase to suggest changes that improve the quality of the software system. Additionally, we posit that the IDEs (equipped with a massive number of static-analysis based checks), are the ideal place for these agents to live. In this paper, we discuss the challenges and issues related to building FM-based refactoring agents that live within the IDE.**

I. Introduction

Foundation Models (FMs) have been shown to be effective at performing various software engineering tasks such as bug repair [1], [2] and code-generation [3]–[5]. Recently, researchers have also applied FMs in the context of software refactoring [6], [7]. In particular, our work on EM-Assist and MM-Assist shows that FMs are capable of providing refactoring suggestions like an expert developer who intimately knows the code; however, FMs hallucinate. More than half of their suggestions are deeply flawed, as they do not understand the mechanics of refactoring – they simply suggest tokens that nicely co-occur together (as seen in their training data). Some of these suggestions are valuable, but distinguishing them from the irrelevant ones is essential. Even when they identify a valid refactoring opportunity, we have found that FMs are incapable of executing the refactoring change correctly, and break the code more often than not. To handle these massive shortcomings, we employ capabilities from the integrated development environment (IDE) to filter out the FM's hallucinations. Then, we use the IDE once again to carry out the refactoring based on the developer's acceptance. This has proven to be an effective way to generate useful suggestions for developers.

While these works focus on suggesting a single type of refactoring, we have observed that developers often perform different kinds of refactoring changes together in the wild, motivated by multiple reasons [8]. For example, while adding a new feature to the software, developers often extract reusable fragments of code, parametrize variables, or move methods to appropriate classes. Currently, no single tool is capable of suggesting this plethora of refactoring changes. Moreover, the increasing prevalence of AI-generated code, such as Google's report that 25% of their recent code is AI-generated [9], highlights concerns about code quality. Studies show that AI-generated code often requires refactoring [10], which underscores the need for an automated refactoring solution.

To create such a tool, we draw inspiration from recent advances in FM-based agent approaches [1], [2], [11]. We envision building an autonomous AI agent capable of suggesting multiple types of refactorings. This agent would be capable of exploring the code base, executing tools, and generating a series of refactorings (a refactoring plan) that improves the code quality while considering the developer's coding preferences and the programming language's best practices. Furthermore, we envision that this agent be integrated into an IDE, enabling it to leverage the IDE's extensive static analysis APIs (which can be utilized as 'tools' by the agent) – including the ability to correctly carry out refactorings. This integration ensures that developers can interact with the agent seamlessly in their familiar environment.

In this position paper, we aim to highlight the need for a refactoring agent and discuss several challenges in building such an agent. Section II sheds light on our recent work in refactoring assistants and motivates the need for an agent. Section III describes the research challenges in building a refactoring agent, from designing to evaluating the agent.

II. Refactoring Powered by IDE-Integrated FM-based Agents

Our recent work has explored the idea of using Foundation Models (FMs) for refactoring tasks [6], [12]. In particular, our work on EM-Assist uses GPT-3.5 to suggest ways to split up large methods – as chosen by a developer. MM-Assist, suggests moving misplaced methods to appropriate target classes. Both of these tools rely on FM-based components to suggest refactoring opportunities, but validate, and further execute the model's suggestion by using an IDE – IntelliJ IDEA. In both cases, we found that FMs had an astonishing effect of behaving like expert developers and suggested refactorings just accordingly in many cases.

However, despite this impressive performance, we found that FMs hallucinate significantly for such refactoring tasks. Although they are trained on massive amounts of code data, they lack an understanding of the specific mechanics of the refactoring change. In the case of the extract method, FMs suggested extracting code that would require multiple variables to be returned. Such a refactoring is not possible in languages like Java. In the case of Move-Method, FMs displayed a lack of awareness about the visibility level of fields and methods (private, protected, public) and suggested moves that would break the code – accessibility to certain fields or methods would be lost. When asked for Extract-Method opportunities, 75% of FM's suggestions were hallucinations, leading to code that would not compile. When asked for Move-Method opportunities, results were similar – 80% of the FM's suggestions are invalid. This makes sense, as the model requires a project-wide understanding of the code to suggest changes.

Even when they identify a valid refactoring opportunity, we have found that FMs are incapable of refactoring the code correctly, resulting in compilation errors, or failing tests. Often, they break the code by refactoring out-of-scope elements, introducing new behaviours, getting the syntax wrong, or failing to call project-specific code correctly.

To filter out these hallucinations, we used the IDE's static analysis capabilities to check and execute refactoring actions. In MM-Assist, we used the IDE's static analysis in two steps: first, identify if a method can be moved in the first place, and second, if the method can be moved to the suggested target class. Only if both of these criteria pass, we present the suggestion to the developer. Finally, if the developer likes MM-Assist's suggestion, we execute it by triggering the Move-Method workflow inside the IDE – a flow that developers are already familiar with.

Prior works have predominantly focused on suggesting a single type of refactoring at a time. However, in real-world scenarios, developers often perform multiple types of refactorings concurrently [8]. For example, extracting a method so that it can be reused, extracting a class to add more functionality, parametrizing a variable, and extracting and subsequently moving a method to a more appropriate host class. Inspired by this observed behavior, we believe that a tool capable of understanding the codebase comprehensively, much like an expert developer, would be highly valuable. Such a tool should be able to propose a plethora of refactoring changes while considering concurrent refactoring changes. To the best of our knowledge, such a tool does not currently exist. This is where we believe an FM-based agent can excel. This is because an agent can learn about various preferences in the developer's (and project's) coding style while also adhering to language-level best practices. Developer preferences could be inferred by analyzing the existing codebase (e.g., naming conventions) or by reviewing which suggestions were previously accepted or rejected. An agents that understands this 'vocabulary', and thinks like another developer in the project, would produce suggestions that are very useful.

Based on our experience in previous works, we cannot trust the FM to execute any changes to the code, as hallucinations often break the code. This is where we believe that the IDE plays a crucial role. Exposing the IDE's static analysis APIs to the agent as 'tools', ensures that changes are executed safely.

III. Research Challenges

We identify four research challenges to build and evaluate a FM-based refactoring agent.

A. Aligning with Developers' Refactoring-Preferences

Developers employ varying criteria for determining when and how to refactor code. Theoretically, cleaner code does not necessarily yield practical benefits in every project context. For example, some developers prefer not to split up long methods if reuse is not promoted (violating the single responsibility principle). Projects may also have their own naming conventions, with developers preferring standard names over more verbose, easier-to-understand names. Projects also have standards for coding style and structure, which are enforced in the code review process. These standards may or may not be explicitly defined in documentation. Then, there are language-wide best practices most developers would choose to adopt.

For an FM-based refactoring agent to be effective, it must be capable of adapting to these project-specific standards. This requires the agent to autonomously learn implicit developer preferences through analysis of the codebase, enabling the generation of suggestions that align with established practices and developer expectations.

B. Agent and Developer Interface

We believe that it is vital to keep the developer in the loop while suggesting refactoring changes. In order to do so, the following research questions emerge: How should refactoring suggestions be presented to the developer? How often should the agent report back its suggestions to the developer? Should the agent autonomously execute changes while allowing the developer to selectively undo modifications, or should it present a curated list of proposed changes for review and approval?

Additional complexity arises when considering the fact that some refactoring changes may be dependent on other changes being executed – a partial ordering. This idea, which we term a 'refactoring plan,' involves executing modifications in a specific sequence, where some changes enable or validate others. Conversely, certain actions may invalidate previous suggestions. Effectively conveying this dynamic relationship to developers is a significant challenge.

Further, there are also questions about prioritizing different refactoring suggestions based on their severity and impact.

C. Sandboxing the IDE's APIs

The reliability of FMs in making direct modifications to source code remains limited, necessitating safeguards to ensure correctness. We envision exposing the IDE's static-analysis APIs to the agent as tools, making sure changes are executed safely. However, simply exposing all the APIs as tools would not work for multiple reasons. There are too many APIs, many of which have overlapping functionality and some of which are irrelevant in the refactoring context. Additionally, exposing all APIs to the agent risks unwanted scenarios, such as deleting untracked files in the project. Like other researchers [11], we also believe that sandboxing relevant APIs is the way to go. This gives tool builders control of the APIs and exposes simple endpoints to the FMs. Further, retrieving the most relevant APIs based on the agent's state would prevent the FM from calling irrelevant functions. For example, when an agent identifies duplicate code suitable for extraction, providing it with tools to perform 'Extract-Method' and 'Parametrize-Variable' would be appropriate.

D. Evaluating the Agent

Assessing the utility of an FM-based refactoring agent presents several methodological challenges. We propose two complementary strategies: (1) comparing the agent's suggestions against an oracle of refactorings derived from open-source projects, and (2) submitting patches to open-source projects to evaluate community acceptance rates. Each of these approaches has distinct challenges that need to be addressed.

1) Evaluation against an Oracle of refactorings in OSS: Mining a dataset of refactoring changes from open-source is possible using a tool such as RefactoringMiner [13]. But, building this oracle is difficult, as most refactoring changes are not done in a stand-alone fashion – they come along with feature additions or bug fixes. Our vision is to build a dataset by constructing a version of the code on top of the developers' changes, but modulo refactoring changes. One way to achieve this is by rolling back refactoring changes on the latest version of the code.

Additionally, since FMs have been trained on internet-wide data, including the entirety of GitHub, they may have seen some of the code in the oracle being transformed as they were, effectively memorizing them. To deal with this issue, we envision creating the dataset only from commits made after the FM's training cut-off date. Doing so ensures that our evaluation truly tests the model's capabilities.

Once this dataset is built, we think that measuring recall is an appropriate performance metric. Measuring precision is not worthwhile as there may be other refactoring opportunities that the developer would indeed perform, but they did not for several reasons. We envision computing recall in two steps: first, checking if the refactoring was identified, and second, verifying if it matches the developer's implementation.

2) Sending Patches to OSS projects: This is the ideal way to measure the usefulness of a refactoring system. However, sending fresh patches to code committed a long time ago may be futile – as developers are often reluctant to change old code, which is considered frozen. Additionally, a PR with simple refactoring changes incurs the entire cost of code review, which may not be worthwhile for small projects. We think that participating in the code review process is a great idea, as this is when developers are most receptive to feedback about newly added code.

IV. Conclusion

Developing an FM-based refactoring agent presents several challenges, including aligning with developer preferences, creating effective interaction interfaces, safely sandboxing APIs, and evaluating agent performance. Addressing these challenges will enable seamless integration of intelligent refactoring tools into development workflows, ultimately enhancing code quality, maintainability, and developer productivity. We encourage the software engineering research community to engage with these challenges and help shape the future of intelligent refactoring tools, ultimately contributing to more efficient and maintainable software systems.

References

[1] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury, "Autocoderover: Autonomous program improvement," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 1592–1604.

[2] I. Bouzenia, P. Devanbu, and M. Pradel, "Repairagent: An autonomous, llm-based agent for program repair," 2025.

[3] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*. Online: Association for Computational Linguistics, Nov. 2020, pp. 1536–1547. [Online]. Available: https://aclanthology.org/2020.findings-emnlp.139

[4] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Online and Punta Cana, Dominican Republic: Association for Computational Linguistics, Nov. 2021, pp. 8696–8708. [Online]. Available: https://aclanthology.org/2021.emnlp-main.685

[5] Github, "Copilot," https://github.com/features/copilot, accessed: 2024-10-07.

[6] D. Pomian, A. Bellur, M. Dilhara, Z. Kurbatova, E. Bogomolov, A. Sokolov, T. Bryksin, and D. Dig, "Em-assist: Safe automated extractmethod refactoring with llms," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, ser. FSE 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 582–586. [Online]. Available: https://doi.org/10.1145/3663529.3663803

[7] A. Shirafuji, Y. Oda, J. Suzuki, M. Morishita, and Y. Watanobe, "Refactoring programs using large language models with few-shot examples," in *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2023, pp. 151–160.

[8] D. Silva, N. Tsantalis, and M. T. Valente, "Why we refactor? confessions of github contributors," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 858–870. [Online]. Available: https://doi.org/10.1145/2950290.2950305

[9] T. Verge, *More than a quarter of new code at Google is generated by AI*, Accessed 2024. [Online]. Available: https://www.theverge.com/2024/10/29/24282757/google-new-code-generated-ai-q3-2024

[10] M. L. Siddiq, L. Roney, J. Zhang, and J. C. D. S. Santos, "Quality assessment of chatgpt generated code and their use by developers," in *Proceedings of the 21st International Conference on Mining Software Repositories*, 2024, pp. 152–156.

[11] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, "Swe-agent: Agent-computer interfaces enable automated software engineering," 2024. [Online]. Available: https://arxiv.org/abs/2405.15793

[12] "Together We Are Better: LLM, IDE and Semantic Embedding to Assist Move Method Refactoring," https://mm-assist.netlify.app/, under review at a SE conference.

[13] N. Tsantalis, A. Ketkar, and D. Dig, "Refactoringminer 2.0," *IEEE Transactions on Software Engineering*, 2022.