Next-Generation Refactoring: Combining LLM Insights and IDE Capabilities for Extract Method

Dorin Pomian*, Abhiram Bellur*, Malinda Dilhara, Zarina Kurbatova
University of Colorado Boulder, USA, *JetBrains Research, Serbia*
dorin.pomian@colorado.edu, abhiram.bellur@colorado.edu, malinda.malwala@colorado.edu, zarina.kurbatova@jetbrains.com
Egor Bogomolov, Timofey Bryksin, Danny Dig
JetBrains Research, the Netherlands, Cyprus, *University of Colorado Boulder, USA*
egor.bogomolov@jetbrains.com, timofey.bryksin@jetbrains.com, danny.dig@colorado.edu
* These authors contributed equally to this work.

***Abstract*—Long methods that encapsulate multiple responsibilities within a single method are challenging to maintain. Choosing which statements to extract into new methods has been the target of many research tools. Despite steady improvements, these tools often fail to generate refactorings that align with developers' preferences and acceptance criteria. Given that Large Language Models (LLMs) have been trained on large code corpora, if we harness their familiarity with the way developers form functions, we could suggest refactorings that developers are likely to accept.**

In this paper, we advance the science and practice of refactoring by synergistically combining the insights of LLMs with the power of IDEs to perform Extract Method (*EM*). Our formative study on 1752 *EM* scenarios revealed that LLMs are very effective for giving expert suggestions, yet they are unreliable: up to 76.3% of the suggestions are *hallucinations*. We designed a novel approach that removes hallucinations from the candidates suggested by LLMs, then further enhances and ranks suggestions based on static analysis techniques from program slicing, and finally leverages the IDE to execute refactorings correctly. We implemented this approach in an IntelliJ IDEA plugin called *EM-Assist*. We empirically evaluated *EM-Assist* on a diverse corpus that replicates 1752 actual refactorings from open-source projects. We found that *EM-Assist* outperforms previous state of the art tools: *EM-Assist* suggests the developer-performed refactoring in 53.4% of cases, improving over the recall rate of 39.4% for previous best-in-class tools. Furthermore, we conducted firehouse surveys with 16 industrial developers and suggested refactorings on their recent commits. 81.3% of them agreed with the recommendations provided by *EM-Assist*. This shows the usefulness of our approach and ushers us into a new era when LLMs become effective AI assistants for refactoring.

***Index Terms*—Refactoring, LLMs, Code smells, Long Methods, Java, Kotlin**

I. Introduction

Excessively long methods that encapsulate multiple responsibilities within a single method are challenging to comprehend, debug, reuse, and maintain [1]–[3]. To mitigate these issues, software developers use Extract Method (*EM*) refactoring – a hallmark refactoring [3] supported by all modern IDEs. This refactoring involves moving a block of statements from a host method to a brand new method, passing the used variables as parameters to the new method, and adding a call to the new method from the host method. This refactoring is reported [4]–[6] as being among the top-5 most frequently performed in practice, both for manual and automated refactoring.

The process of performing an *EM* consists of two phases: (i) choosing the statements to extract and (ii) applying the mechanics to perform this refactoring. While the application part has been a staple feature of all modern IDEs, they leave it up to developers to choose the statements to extract. Researchers developed several tools that suggest which statements to extract based on static analysis [1], [7]–[10], or machine learning (ML) models [11], [12]. These tools use software quality metrics (e.g., coupling, cohesion), which they optimize based on heuristics or by training ML classifiers.

While these tools adhere to software engineering principles like the Single Responsibility Principle [13], they fail to align with real-world *EM* instances. We posit that the decision to refactor is both *a science and an art*. Developers use both their knowledge of software engineering principles *and* rely on their own experience and subjective interpretation of the code context and what makes a good method. This might explain why developers are reluctant to use automated refactorings [14] and the large gap between high metrics scores and their low acceptance by developers [15]–[20].

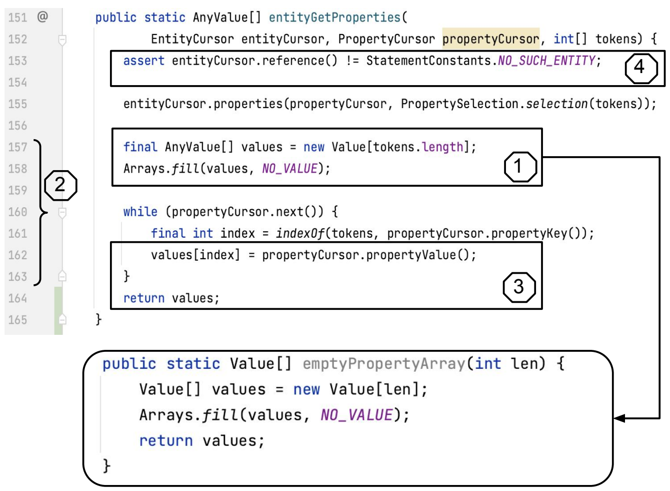
Recently, Large Language Models (LLMs) [21]–[23] are emerging as powerful companions for several software engineering tasks [21], [24]–[27]. Given LLMs' training on extensive code repositories that contain billions of methods written by actual developers, we hypothesize that they are more likely to imitate human behavior by mimicking how developers form functions. Thus, they are likely to suggest refactorings that developers would accept. Our formative study on 1752 *EM* scenarios from 12 open-source projects revealed that LLMs are very effective in giving expert suggestions. Moreover, they are very prolific, producing on average 27 suggestions per method. However, we also discovered that LLMs are unreliable: 76.3% of their suggestions are *hallucinations*, i.e., they seem plausible at first, but are actually deeply flawed. We found that 57.4% of the suggested refactorings contain code fragments that are invalid to extract (e.g., would produce compile errors), and 18.9% are not useful (e.g., suggest to extract the whole host method).

To advance the state of the art and practice for refactoring, we bridge several important gaps. First, we bridge the gap between the suggestion of refactorings and developers' actual practices by grouping statements into methods that resemble human-written code. Second, we bridge the gap between suggesting and applying refactorings by supporting the whole end-to-end process in a way that provides maximum automation while taking into account human input.

We have designed, implemented, and evaluated *EM-Assist*, an IntelliJ IDEA plugin, that supports Java and Kotlin *EM* refactorings. *EM-Assist* synergistically combines (i) the creative capabilities of LLMs, (ii) static analysis techniques to filter and enhance LLM-provided suggestions, and (iii) the full power of a state-of-the-practice commercial IDE to apply refactorings safely. *EM-Assist* first repeatedly prompts the LLM in a few-shot learning style to generate a diverse range of refactoring suggestions. Subsequently, it eliminates two kinds of hallucinations: (i) it employs static analysis techniques from the IDE to eliminate invalid suggestions (i.e., illegal groupings of code statements), and (ii) it eliminates suggestions that are not useful (e.g., that include the whole body of the host method, or one single line). Then it further enhances the remaining valid suggestions based on program slicing techniques. Following this step, it ranks and prioritizes high-quality suggestions so that it would not overwhelm the developer with too many suggestions. It then presents the ranked suggestions to the developers. Lastly, it encapsulates the user-chosen candidate into a refactoring command and uses the IDE to correctly execute the refactoring.

We designed a comprehensive empirical evaluation to determine the benefits of our novel approach, the pros and cons of using LLMs, and a *sensitivity analysis*. To determine the effectiveness of *EM-Assist*, we use two complementary methods. First, we use a publicly available corpus [11] of 122 *EM*s that other researchers used in the past. The results show that our *EM-Assist* outperforms state-of-the-art static-analysis tools such as JDeodorant [1], JExtract [28], SEMI [8], LiveRef [29], [30], and also outperforms ML-based techniques like REMS [12] and GEMS [11]. *EM-Assist* correctly suggests the ground truth *EM* refactoring among the top-5 candidates 60.6% of times, compared to 54.2% reported by existing ML models, and 52.2% reported by existing static analysis tools. Second, we significantly expanded the previous community corpus of 122 *EM*s with a diverse corpus of 1752 *actual EM* instances from open-source projects. *EM-Assist* correctly suggests the actual refactoring performed by the developers in 53.4% of cases, an improvement over the 39.4% rate of the previous best-in-class static analysis tool.

Moreover, to assess whether refactoring suggestions generated by *EM-Assist* are useful, we employed firehouse surveys [31] with 16 industrial developers from a reputable software company. We presented them with refactoring suggestions for lengthy methods they previously committed into their software repository. 81.3% of respondents agreed that the suggestions were beneficial and suitable for application in their codebase. They wish to see *EM-Assist* in production in IntelliJ IDEA and use it in their daily coding. This demonstrates that *EM-Assist* generates suggestions that are highly likely to be embraced by developers and ushers us into a new era of refactoring when developers are accepting AI assistants.



```
151 @      public static AnyValue[] entityGetProperties(
152                EntityCursor entityCursor, PropertyCursor propertyCursor, int[] tokens) {
153            assert entityCursor.reference() != StatementConstants.NO_SUCH_ENTITY;
154
155            entityCursor.properties(propertyCursor, PropertySelection.selection(tokens));
156
157            final AnyValue[] values = new Value[tokens.length];
158            Arrays.fill(values, NO_VALUE);
159
160            while (propertyCursor.next()) {
161                final int index = indexOf(tokens, propertyCursor.propertyKey());
162                values[index] = propertyCursor.propertyValue();
163            }
164            return values;
165        }
```

```
public static Value[] emptyPropertyArray(int len) {
    Value[] values = new Value[len];
    Arrays.fill(values, NO_VALUE);
    return values;
}
```

Fig. 1: The numbered code snippets represent (1) an extract function refactoring in the project Neo4j, commit a05a8c5, (2) a suggestion made by static analysis tool, (3) an invalid suggestion from LLM, (4) a not useful suggestion from LLM.

In summary, this paper makes the following contributions:

(1) We present the first approach that automates the full refactoring lifecycle by synergistically combining LLMs and IDEs. Thus, we effectively bridge the gap between refactoring recommendation and application, and we shrink the gap between refactoring suggestions and developer practices.
(2) We discovered best practices for prompting and tuning LLMs to produce effective refactoring suggestions and we reveal LLMs' strengths and weaknesses for refactoring.
(3) We designed, implemented, and evaluated *EM-Assist*, a plugin for IntelliJ IDEA, that integrates all these ideas.
(4) Our comprehensive empirical evaluation on a corpus used by others, as well as significantly extended corpus that replicates 1752 real-world refactorings shows that *EM-Assist* outperforms static analysis-based techniques and ML approaches that suggest *EM* refactoring. Moreover, our survey with 16 industrial developers provides insights about the reasons why developers accept or reject *EM-Assist*'s suggestions.

To aid reproducibility, the source code and all experimental datasets are available on GitHub [32]. *EM-Assist* is freely available to be installed from the JetBrains Marketplace [33]. A demo video is on YouTube [34].

II. Motivating Example

In this section we first illustrate the challenges of suggesting code fragments to extract into new methods in a way that aligns with developer preferences. Then we explain the process of unleashing the full potential of LLMs for *EM* refactoring. Figure 1 shows an *EM* refactoring performed by Neo4J developers. They extracted statements in lines 157 and 158 into a new method named `emptyPropertyArray` (① in Figure 1). This refactoring decision is intuitively sound as these two statements collectively fulfill a single responsibility, namely, the creation of an empty array.

When replicating this real-world scenario, existing techniques [1], [7], [8] for suggesting *EM* would recommend extracting lines 157–163 (② in Figure 1). This recommendation arises from the fact that the variable `values` is used later in the code following the two aforementioned statements. However, this selection of statements does not align with the developers' actually performed refactoring (① in Figure 1) despite the fact that the extraction of lines 157-163 adheres to the static analysis principles (e.g., extracting a program slice) and software-quality metrics (e.g., improving the cohesion of statements in a method) governing existing tools. This finding is in line with research that shows that code-quality metrics do not necessarily capture code quality improvements as perceived by the developers [15], [16].

Next, we used a LLM to suggest *EM* refactorings for this method, specifically opting for GPT-3.5 [35] because it is cost-effective. We provided the LLM with a prompt that included the code snippet in the host method `entityGetProperties`, followed by a description of the *EM* refactoring process and an instruction to generate refactoring suggestions for the method. As LLMs produce results that are non-deterministic, we repeated the same prompting and collected suggestions until we reached a fix-point, i.e., the LLM no longer produced new suggestions. We reached the fix-point in 9 iterations.

The LLM came up with 9 distinct suggestions for which statements to extract, among these including the one that the Neo4J developer performed. In addition to 3 applicable suggestions, the LLM also proposed to extract lines 162–164 (③ in Figure 1). If we did so, the code would not even compile because the suggestion starts inside the `while` loop and goes past the closing bracket. We call these suggestions *invalid*. The LLM produced 3 invalid suggestions. While the LLM might have seen those code statements being associated together in codebases, because the LLM does not fully understand the semantics of *EM* and syntactical rules of well-formed code, those associations render the extracted method not compilable.

Moreover, the LLM suggested extracting the code fragment in line 153 (④ in Figure 1), which is a one-liner. While it is possible to extract this line correctly, we consider it *not-useful* as it is improbable for developers to undertake this suggestion. Similarly, the LLM suggested to extract the whole method body into a new method, which does not provide any value for the developers. We call such suggestions *not-useful*. In this experiment, the LLM produced 3 not-useful suggestions. The invalid suggestions, together with the not-useful ones, represent *hallucinations* of the LLM.

This experiment showcases the strengths of LLMs: being prolific in generating 9 suggestions, among those is 1 suggestion that aligns with the way how the developers actually performed the refactoring. However, it also showcases the LLM's weakness: it produced 6 hallucinations, of which 3 are invalid suggestions and 3 are not useful. This shows that it is imperative to exercise caution and not exclusively rely on LLM-generated suggestions when conducting refactorings assisted by LLMs. Moreover, it shows that while LLMs have a huge potential to come up with refactorings that match human's acceptance criteria, developers that embark on such a journey need to work hard to tap into such potential: they need to tame LLMs non-determinism by repeated prompting (9 in our example), collect those 9 suggestions, and sift through and discard 6 hallucinations. Moreover, as we show in Section IV-D, developers might need to repeat the whole suite of experiments for different LLM "temperature" settings: these control the level of variability for the solutions produced by the LLM. Furthermore, developers might want to use various LLMs (in Section IV-C we tried three LLMs), altogether resulting in a combinatorial explosion of experiments.

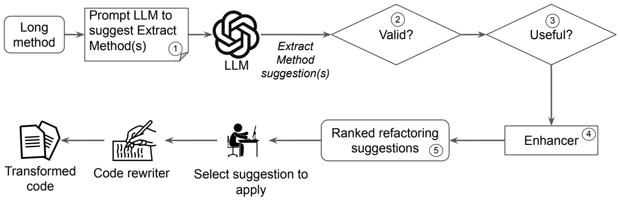

Long method
Prompt LLM to suggest Extract Method(s)
1
LLM
Extract Method suggestion(s)
2
Valid?
3
Useful?
Enhancer
4
Ranked refactoring suggestions
5
Select suggestion to apply
Code rewriter
Transformed code

Fig. 2: The workflow of generating refactoring suggestions and then applying them with *EM-Assist*.

This is where our tool, *EM-Assist*, liberates the developers so they can focus on the creative part: using their expertise to examine a small number of high-quality suggestions.

III. Technique

In this section, we present the workflow that our novel approach and tool, *EM-Assist*, uses to automatically suggest and perform *EM* refactoring in Java and Kotlin codebases.

Figure 2 shows the architecture and the steps performed by *EM-Assist*. First, *EM-Assist* invokes the LLM (① in Figure 2) by prompting it to generate *EM* suggestions for the selected method (see Section III-A). Next, our algorithm removes hallucinations and retains *applicable* refactoring options that are free of errors (② in Figure 2, detailed in Section III-B) and are useful (③ in Figure 2, detailed in Section III-C). These steps play a key role in retaining suggestions that align with the way developers extract code, while simultaneously mitigating the risk of introducing bugs or non-compilable code.

Next, our algorithm further *enhances* (④ in Figure 2, detailed in Section III-D) the quality of the suggestions by expanding or shrinking the code fragment in the suggestion based on program slicing rules. These rules ensure that refactorings better align with the developer's intentions. Then, the tool ranks the refactoring suggestions (⑤ in Figure 2, detailed in Section III-E) to prevent overwhelming the developer, presenting only the top-n ranked options for the developer to choose from. Tool's GUI displays the suggestions and shows a preview of the signature of the extracted method, along with the code fragment that would be extracted. Based on the user's decision, *EM-Assist* encapsulates the suggestion in a refactoring command and invokes IntelliJ IDEA to perform the refactoring correctly.

Definition III.1. (Long method) is chosen by a developer for extract method refactoring, denoted as $l_i(s_i, e_i)$. The method starts at line s_i, where its signature is defined, and ends at line e_i containing the closing curly bracket that signifies the end of the method declaration. The method's length, L_i, is $e_i - s_i$.

Definition III.2. (Extract Method) denoted as $e_j(n^e_j, (s^e_j, e^e_j))$, where n^e_j represents the method name, (s^e_j, e^e_j) signifies a code fragment within a long method. It is identified by the starting line number, defined as $s^e_j > s_i$, and the ending line number, defined as $e^e_j < e_i$. These line numbers are calculated with respect to the original source code file in which the host method $l_i(s_i, e_i)$ is located.

Definition III.3. (Extract Method Suggestions ($\mathcal{S}$)) is a set of extract method suggestions generated by an LLM for a long method l_m, formally, $\mathcal{S} = \{e_i(n_i, (s^e_i, e^e_i)), e_j(n_j, (s^j_s, e^j_e)), \ldots\}$

Definition III.4. (Invalid Extract Methods ($\mathcal{I}$)) is a subset of the $\mathcal{S}$ suggestions, such that it does not satisfy the validity conditions ($C(e_i)$), ensuring that the resulting code remains compilable. Formally, $\mathcal{I}_i = \{e_i \in \mathcal{S} \mid \neg C(e_i)\}$. The remaining set, $\mathcal{V}$, comprises suggestions that satisfy the validity conditions, making them valid suggestions for use. Formally, $\mathcal{V}_i = \{e_i \in \mathcal{S} \mid C(e_i)\}$.

Definition III.5. (Not Useful Extract Methods ($\mathcal{NU}$)) is a subset of $\mathcal{V}_i$ comprising elements that fail to meet the criteria for usefulness, $U(e_i)$. This subset ensures that the suggestions are neither too large, encompassing almost the entire method body, nor too small, essentially one-liners. Formally, $\mathcal{NU}_i = \{e_i \in \mathcal{V} \mid \neg U(e_i)\}$. Conversely, useful suggestions $\mathcal{U}_i$, consist of elements satisfying the usefulness criteria $U(e_i)$, suitable for method extraction, defined as $\mathcal{U}_i = \{e_i \in \mathcal{V} \mid U(e_i)\}$.

A. Generating EM Suggestions

We utilize LLMs to generate *EM* recommendations for a given input long method. In this section, we delve into the preparation of LLM for *EM* refactoring suggestions, as well as the parameters of the LLM that require tuning.

*1) **Prompt engineering**:* LLMs are versatile models capable of various applications, but they require specific preparation when used for a particular task [26]. To facilitate this, we employ *in-context learning* [36], where we provide all the instructions needed to solve the task right in the model's input, including task definition and relevant context information. To further enrich the prompt, we employ few-shot learning [35], [37] by incorporating a set of 2 examples into the prompt, following best practices as discussed by Gao et al. [38]. The prompt includes: (i) A succinct overview of *EM* refactoring, (ii) The definition of a long method, (iii) A JavaDoc string when available, (iv) 2 examples of refactoring for one long and one short method, and (v) Precise instructions for the desired output format. An example prompt is accessible on our companion website [39].

*2) **Generating an extensive array of suggestions**:* The output of an LLM depends on two factors: (i) the internal variable "Temperature" (T), and (ii) the number of iterations (I) [35], [37], [40], which indicates how many times the same prompts are inputted. Temperature serves as a regulator for the model's output randomness. Higher values, such as 0.9 or 1.2, produce more diverse and unpredictable outputs, whereas values closer to zero produce focused and less diverse results. We determine the optimal value empirically (see Section IV-D).

Even when identical prompts are presented multiple times, the output can exhibit variations due to: (i) LLMs employ a combination of learned patterns and random sampling in generating responses, introducing slight deviations in their answers each time due to inherent randomness; (ii) LLMs possess the capability to explore various potential solutions and refine their responses iteratively through interactions and feedback [35], [37], continually enhancing their outputs in subsequent iterations. Therefore, the quality and quantity of predictions made by an LLM are influenced by the frequency of prompting the same prompt.

In Section IV-D, we conduct a *sensitivity analysis* to determine how the performance of *EM-Assist* varies with each combination of temperature and iterations. This analysis enables *EM-Assist* to use the optimal settings for generating refactoring suggestions.

B. Removing Invalid EM Suggestions

Not all suggestions generated by LLMs can be directly applied to codebases, as some may lead to non-compilable code. Therefore, *EM-Assist* analyzes each suggestion to ensure it can be extracted without breaking the code's scope. For example, suggestion ③ in our motivating example (Figure 1) fails the scope analysis, and *EM-Assist* expands the code fragment of the suggestion so that both the start and end lines have the same scoping level: the new suggestion's code fragment starts from line 160, while the end line stays the same. This change, however, leads to a compilation error.

To remove suggestions that might lead to non-compilable code, *EM-Assist* uses the rules below:

(i) *Variable Usage:* This criterion ensures that all necessary variables and objects are accessible within the same scope or are provided as parameters, maintaining the code's self-containment.

(ii) *Return Values:* If the code being extracted produces a result or modifies the state of objects, *EM-Assist* validates that the return value and side effects are appropriately handled. For example, it checks if the return value is used subsequently in the host method or if any thrown exceptions are caught.

(iii) *Number of Return Values:* Java methods can return only one value, and hence, *EM-Assist* validates whether the code fragment to be extracted might produce more than one result, in which case it discards the suggestion.

(iv) *Control Flow: EM-Assist* reasons about the control flow within the extraction suggestion to ensure that it can be extracted as a separate method without causing logical errors. For example, it checks for any early returns or breaks that might affect the control flow and would render the extracted method to have a different behavior than the original code.

EM-Assist checks all these conditions through the static analysis infrastructure in the IntelliJ Platform[1] (the open-source platform IntelliJ IDEA and other JetBrains IDEs are built upon) that verifies refactoring preconditions. As LLMs are not aware of the semantics of *EM* refactoring, but only about the code tokens that are associated together in its training data, this step is crucial for discarding a large number of invalid suggestions (57.4% of all suggestions).

C. Removing Refactoring Suggestions That Are Not Useful

Once we retain only valid refactoring suggestions, the next step filters out those suggestions that are not useful and retains the ones that are applicable and likely to be performed by a developer. To identify suggestions that are not useful, we employ two rules: (i) Exclude *EM* suggestions that encompass 88% or more of the lines present in the original host method. We determined this threshold empirically as we found that that doc-strings, method-signatures and blank-lines account for a small percentage of the method's size. (ii) Following practices adopted by other researchers [1], [29], [41], [42], we exclude *EM* suggestions that are one-liners.

Extracting an entire method or a one-liner into another method is, in general, not useful for the purpose of improving existing code. These might be useful if the developers plan to add new code that implements new features into the host method or into the extracted method. Notice that we designed *EM-Assist* for *code renovation* and not for feature expansion.

D. Enhancing Refactoring Suggestions

After identifying useful suggestions (Definition III.5) devoid of hallucinations, the next step enhances these suggestions to further improve the quality of valid suggestions.

(i) *Program slicing*: Taking inspiration from tools that rely solely on static analysis, such as those utilizing program slicing [1], [7], [8], [43], [44], we leverage program slicing to augment refactoring suggestions provided by LLMs. We use rules that better align with developer preferences to avoid creating small methods with several arguments.

Let EF_s be the initial statement of the code fragment to be extracted into a new method, with EF_{s+i} indicating statements below EF_s and EF_{s-i} denoting statements above EF_s within the long method. Let EF_{s-1} represent the variable declaration statement for variable v, and this variable is used inside the code fragment of the suggested method extraction, identified through live variable analysis based on a def-use chain. In such cases, we increase the scope of the code to be extracted by one statement, including EF_{s-1} in the extracted method, and we adjust the starting line of the suggestion to match the starting line of EF_{s-1}. This avoids having to pass v as a parameter.

(ii) *Control statements*: If the code fragment in the extracted method starts with a control statement (i.e., EF_s is the check block of an `if` statement), we shrink the code fragment to start at EF_{s+1} and contain only the block of the `if`. This leaves the check condition intact in the host method so that the developer can see under which conditions the would-be extracted method is called, thus making the code easier to read.

We evaluate the effectiveness of these heuristics in Section IV-D. We plan to experiment with other heuristics based on the feedback we receive on JetBrains Marketplace [33].

E. Ranking Refactoring Suggestions

The number of the remaining applicable suggestions per host method can be large (on average 10 per method) and overwhelm the developer. To prioritize high-quality suggestions, we implemented a ranking mechanism that gives precedence to recommendations consistently identified by the LLM. This approach elevates frequently highlighted suggestions, employing the LLM's extensive knowledge of coding best practices. Next we present our ranking function.

$$\mathcal{T}_1(e_i) = \mathcal{H}(e_i) = \sum_{i=1}^{N} \mathcal{F}(line_i) \tag{1}$$

$$\mathcal{T}_2(e_i) = \mathcal{P}(e_i) \tag{2}$$

$$\mathcal{T}_3(e_i) = \mathcal{H}(e_i) \cdot \mathcal{P}(e_i) \tag{3}$$

Where:

N :Total number of lines in the *EM* suggestion.

$\mathcal{F}(line_i)$:Number of times the i^{th} line appears in all the suggestions.

$\mathcal{H}(e_i)$:Heat of the *EM* suggestion e_i.

$\mathcal{P}(e_i)$:Popularity of the *EM* suggestion e_i.

Our approach begins with $\mathcal{T}_1(e_i)$, where we assign scores to each suggestion according to a "heat map" of the host method. Intuitively, this ranking measures the LLM's confidence that a certain line of code in the host method belongs to another method. To compute the heat map, we record how many times each line appears in all applicable suggestions. If a line is absent from all suggestions, we assign it a score of zero. We repeat this procedure for every line within the host method. Then, we aggregate individual line scores to determine each suggestion's overall heat score, comprising multiple code lines, and then rank these suggestions based on their heat scores.

$\mathcal{T}_2(e_i)$ ranks suggestions by considering their popularity. As discussed in Section III-A2, we re-prompt the LLM several times with the same prompt. While we remove duplicated suggestions, we keep track of how many times a certain suggestion is produced by the LLM during re-prompting. Given that LLMs have been trained on extensive codebases enabling them to mimic how real developers construct methods, we give precedence to suggestions that appear repeatedly.

Finally, $\mathcal{T}_3(e_i)$ combines both the heat and popularity of suggestions using a weighted average, aiming to strike a balance between the importance of these two factors. We evaluate the effectiveness of the rankings in Section IV-D.

IV. Evaluation

We empirically evaluate *EM-Assist* by answering the following research questions:

RQ1. How effective are LLMs at generating refactoring suggestions? Understanding the capability of LLMs to produce *EM* suggestions is essential for our tool's effectiveness.

[1] The IntelliJ Platform: https://www.jetbrains.com/opensource/idea/.

Therefore, we conducted a quantitative analysis involving three of the latest LLMs available.

RQ2. How do refactoring suggestions change with different LLM parameters? This is important for integrating LLMs into tools and for researchers using LLMs for refactoring. We conduct a *sensitivity analysis* to assess the quality of these suggestions, focusing on the recall rate for the top-5 suggestions (Recall@5) across various LLM parameters, and examine the effects of our enhancements and ranking methods.

RQ3. How effective is *EM-Assist* in providing refactoring recommendations over existing approaches? To quantitatively evaluate this, we conducted a baseline comparison with 6 other tools that employ static analysis or machine learning techniques to suggest *EM* refactorings.

RQ4. How useful are the provided recommendations to developers? *EM-Assist* is a code renovation tool that developers use interactively. Thus, it is important to evaluate whether *EM-Assist* makes suggestions that developers accept. We employ firehouse surveys with professional developers from our collaborating enterprises, focusing on newly created long methods they committed into code repositories.

A. Datasets

To answer our research questions, we use two datasets: the "Community Corpora" previously used by other researchers working in this field, and the "Extended Corpus" which we released to address the limitations of previous corpora and to increase its size.

(i) ***Community Corpus:*** consists of 122 Java methods and their corresponding *EM* refactorings using five open-source repositories: MyWebMart, SelfPlanner, WikiDev, JHotDraw, and JUnit. This dataset previously served as the foundation for evaluating various state-of-the-art *EM* refactoring automation tools, including *JExtract* [28], *JDeodorant* [1], *SEMI* [8], *GEMS* [11], and *REMS* [12]. The Community Corpus, although extensively used by researchers, can be seen as subjective. It combines (i) theoretical refactoring scenarios assessed by third-party (usually students) who are not familiar with the open-source projects, and (ii) synthetic refactorings devised by corpus designers, based on inlining existing methods followed by extracting these same methods from the expanded host methods. While these refactorings are realistic, they are not actual refactorings that developers performed. Thus, we created another corpora that addresses these limitations.

(ii) ***Extended Corpus:*** To further strengthen our evaluation's robustness with a broad oracle of actual refactorings by developers and ensure more generalizable results, we constructed Extended Corpus. To create it, we employed RefactoringMiner [6], the state-of-the-art tool for mining refactorings from commits, with a reported precision of 99.8% and recall of 95.8% for detecting *EM*. We ran RefactoringMiner on 12 open-source repositories, including notable projects like CoreNLP, Guava, and Gson, which span a wide range of domains such as machine learning, database management, data handling, and web technologies. We filtered out refactoring commits that include one-liners and extracted methods whose bodies overlapped significantly with the host method. Additionally, we filtered out non-automatable [45] refactorings, such as those entailing feature additions. For this purpose, we leveraged the state of the practice IDE for *EM*, IntelliJ IDEA, using start and end line numbers of the extracted code fragment provided by RefactoringMiner to determine whether IntelliJ IDEA could automatically extract the code fragment identified.

Consequently, we retained 1752 *EM*s from these repositories. We demonstrate that this corpus is representative in terms of the length of the host method, cyclomatic complexity [46] of the host method, and Halstead metric difficulty [47] of the host method, as shown in Figure 3.

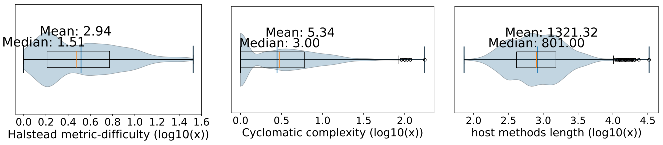


Fig. 3: Metrics for host methods in our corpus

B. Evaluation Metric: Recall@n with m% tolerance

Previous researchers [1], [11], [12], [28] have designed metrics to evaluate *EM* suggestion tools. In their evaluations, they analyze the top-n ranked suggestions generated by their tools at various $m\%$ tolerance levels. We also employ *Recall@n with $m\%$ tolerance* to evaluate *EM-Assist*. First, to explain $m\%$ tolerance level, let us examine a specific instance. Consider an extract method suggestion $e_i(n_i, (s_i, e_i))$ where s_i and e_i are the start and end line numbers of the suggested code fragment to be extracted, while the oracle specifies $e_j(n_j, (s_j, e_j))$ as the actual refactoring. The $m\%$ tolerance verifies whether the start and end line numbers of the suggestion are within the start and end lines of the ground truth extracted method, with a tolerance threshold at most $m\%$ of the length of the ground truth method, L_i. Formally, $|e_i - e_j| + |s_i - s_j| \leq n/100 * L_i$. Next, Recall@n is calculated as the percentage of host methods for which at least one of the top-n *EM* suggestions matches the refactoring specified in the oracle, given $m\%$ tolerance level.

Given the non-deterministic nature of LLM outputs, recall rates for *EM-Assist* may vary across invocations. However, the impact of this on *EM-Assist* is minimal since we invoke the LLM multiple times. Despite the minimal effect, during the experiments, when we report the recall rate for *EM-Assist*, we repeat the experiments $\mathcal{N}$ times and consider a distribution of recall rates, then report the mean and standard deviation of these values. We perform $\mathcal{N}$ experiments by bootstrapping our data [48]. To understand the performance at the ith iteration, we take $\mathcal{N}$ random samples from our dataset of I iterations. While a larger $\mathcal{N}$ is preferable, we use $\mathcal{N}$ = 30, following the best practices suggested by Arcuri and Briand [49].

*C. **RQ1: Effectiveness of LLMs***

Numerous LLMs have been developed by both open-source communities and proprietary companies [22], [23],

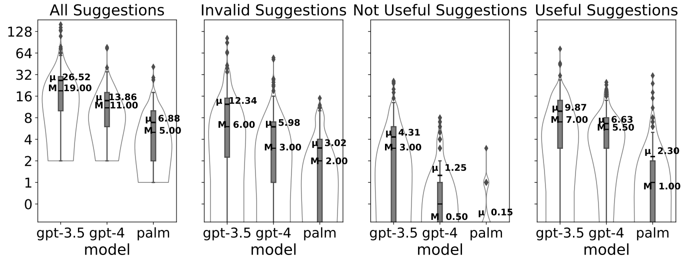


Fig. 4: The capabilities of LLMs in generating refactoring suggestions. The plots show the number of suggestions per host method (notice the exponential scale)

[50]–[53]. Among them, (i) PaLM[2] [23], (ii) GPT-3.5 [35], and (iii) GPT-4 [22] are well-known and the largest LLMs, developed by Google and OpenAI, and we use them for the experiments. While these models were not developed specifically for refactoring, they are versatile. We determine their effectiveness in generating *EM* refactoring suggestions.

*1) **Subject Systems and Experimental Setup**:* We employed a two-step process to assess the quality of refactoring suggestions generated by LLMs.

Step 1: We used Community Corpus and then prompted the LLM with each method to generate refactoring suggestions. To maximize the capabilities of LLMs to generate suggestions, we employed an iterative approach by adjusting the temperature parameter during LLM prompting, as explained in Section III-A2. Specifically, we conducted fixed-point iterations, repeatedly prompting the LLM until it no longer generated new suggestions for each temperature value from the set {0, 0.2, 0.4, 0.6, 0.8, 1.0, 1.2}. This iterative strategy allowed the LLM to create refactorings with varying levels of randomness and expand its search with each subsequent response to the prompt. Then, we analyze the quality of the suggestions by quantitatively studying invalid suggestions (Definition III.4), and not useful suggestions (Definition III.5).

Step 2: To significantly increase the validity of the observations made during Step 1, we employed the Extended Corpus and prompted it with the best-performing LLM parameters from Step 1. We then analyzed the quality of the suggestions, following the same process as in Step 1.

*2) **Results**:* In Figure 4, box and violin plots show the distribution of suggestions per host method. Starting from the left, total suggestions, then those that are invalid, not useful, and finally, the useful suggestions generated by each LLM. The data reveals that all three studied LLMs excel in generating refactoring suggestions, with GPT-3.5, GPT-4, and PaLM averaging 27, 14, and 7 refactoring suggestions per host method, respectively. However, they also produced invalid suggestions with averages of 12, 6, and 3, respectively, and not useful suggestions with averages of 4, 1, and 0, respectively. Notably, GPT-3.5 yielded the highest number of useful suggestions, with an average of 10 per host method, compared to 7 and 2 averages for GPT-4 and PaLM, respectively. These results highlight the effectiveness of LLMs in generating refactoring suggestions while also highlighting the need for additional techniques to select only useful suggestions.

TABLE I: Ablation Study: Mean and Standard Deviation of Recall Values for different stages in *EM-Assist*'s pipeline.

Method	Recall@5 Tolerance 1%	 Tolerance 2%	 Tolerance 3%
Random rank, no heuristics	32.9% ± 2.3	34.2% ± 2.3	37.4% ± 2.4
Random rank, with heuristics	44% ± 2.8	44.8% ± 2.9	49.6% ± 2.9
Ranking and heuristics	**57.6%** ± 1.6	**58.4%** ± 1.6	**63%** ± 1.7

To further bolster the validity of this claim, we selected GPT-3.5, the top-performing LLM in terms of applicable suggestions, and applied it to the Extended Corpus. Our observations, consistent with the findings in Step 1, revealed that GPT-3.5 generated total 22741 refactoring suggestions, of which 17356 were hallucinations (i.e., invalid and non-useful), leaving only 5385 (23.7%) useful suggestions. This underscores the importance of employing post-processing techniques on LLMs' output, they cannot be used as-is for refactoring tasks.

LLMs excel at generating *EM* suggestions, yet only 23.7% of these suggestions are useful.

D. *RQ2 : Sensitivity Analysis of* EM-Assist

We determine the optimal values for LLM hyper-parameters so that we can harness the full potential of LLMs when generating *EM* refactorings. Then we study the contribution of our design decisions on enhancing the refactoring suggestions made by *EM-Assist*. This is crucial for revealing best practices for using LLMs for refactoring tasks, and it also enables us to quantitatively assess *EM-Assist*'s advancements over the raw performance of LLMs.

*1) **Subject Systems and Experimental Setup**:* We use our oracle dataset, Community Corpus, to analyze the output of *EM-Assist* by examining the effects of specific Temperature values ($t \in 0, 0.2, 0.4, 0.6, 0.8, 1.0, 1.2$) and iteration numbers ($i \in 1, 2, \ldots, 10$). For each parameter combination (t, i), we invoke *EM-Assist* to generate extract method suggestions for a host method within the dataset. In line with the methodologies adopted by previous researchers [1], [11], [12], [28] for assessing earlier *EM* automation tools, we evaluate each parameter setting by calculating Recall@5 with a 3% tolerance level. Here, to identify the optimal parameter values (t, i), we analyze a distribution of recall values (as explained in Section IV-B) and select the mean recall. Then, we assess the recall both with and without *EM-Assist*'s enhancements to the LLM suggestions.

*2) **Results**:* Figure 5 shows via heatmaps how *EM-Assist*'s mean recall varies with LLM Temperature/Iterations. The results show that higher temperature values and more iterations consistently yield superior results. Even though LLM randomness increases with higher temperature values, potentially

[2] Since the time we conducted the experiments, Google has improved the performance of its PaLM model, and it was subsequently replaced by Gemini

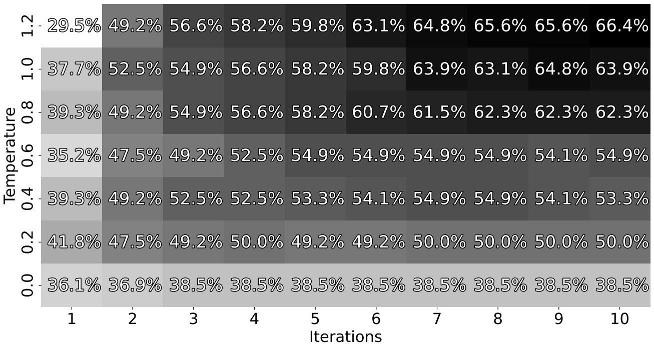


Fig. 5: Change of Recall@5 along with Temperature and iterations for the Community Corpus

leading to a greater variety of suggestions and possibly more hallucinations with more iterations, the enhancements implemented in *EM-Assist* effectively refine and rank these suggestions in such a manner that its recall improves alongside these parameters. Notably, *EM-Assist* recorded its peak Recall@5 scores at 63%, with the configuration set to a temperature of 1.2 and 10 iterations, establishing these parameters as the optimal settings for *EM-Assist*'s applications. Although we expect to see a better recall after running more iterations, we do not deploy our tool with such settings as it is less cost-effective. Additionally, we observed that increasing the temperature beyond a value of 1.2 causes the LLMs to become too random, resulting in completions that did not follow the formatting instructions in the prompt. This made it impractical to automatically parse the LLM output. Hence we do not use a temperature value beyond 1.2.

To further study the contributions of various components within the *EM-Assist*'s pipeline (see Figure 2), we conducted an ablation study. We structured this into three distinct phases: (i) In the initial phase, we processed the raw output from the LLM without any heuristic enhancements and randomly selected five candidates. (ii) We considered the enhanced output with heuristics (see Section III-D) and arbitrarily choose five suggestions after enhancement. (iii) We took into account the enhanced output and applied the tool's ranking mechanism. These stages were instrumental in understanding the individual contributions of the various modules to the overall performance. As shown in Table I, the results emphasized that significant improvements were observed across all tolerance levels, with a notable enhancement at the 3% tolerance level. Specifically, the baseline mean recall of the LLM was recorded at 37.4%. Through our design decisions, this figure was elevated to 63%. This enhancement vividly illustrates the substantial influence of the strategic decisions embedded in our tool's design.

With higher temperature (1.2) and more iterations (10), LLMs produce better suggestions for *EM*s. Our ranking and heuristics further amplified the quality of the suggestions, achieving an uplift of up to 26 percentage points.

E. RQ3 : Effectiveness of EM-Assist

EM-Assist introduces a novel approach for refactoring recommendation that harnesses the power of LLMs. Thus, we assess its effectiveness relative to existing state-of-the-art tools.

1) ***Subject systems and Experimental Setup****:* We selected six tools that are representative of a wide array of techniques, including four (JDeodorant [1], JExtract [28], SEMI [8], and LiveRef [30]) that use static analysis-based rules and software quality metrics, and two (REMS [12] and GEMS [11]) that use prediction models based on machine learning techniques.

We first employ Community Corpus to evaluate the effectiveness of the selected tools. Following practices employed by the authors of other tools, we evaluate top-5 suggestions generated by the tools, calculating mean Recall@5 at tolerance levels of 1%, 2%, and 3% (for details on evaluation metrics refer to Section IV-B). For *EM-Assist*, we compute the metric for the best-performing hyper-parameters. For four tools (GEMS, JDeodorant, JExtract, SEMI), we reuse the evaluation results reported by other researchers [11] because they used the same dataset and they defined the same metric formulation as we do. For two tools (*REMS* and *LiveRef*), we had to run them ourselves: *LiveRef* had not previously been executed on Community Corpus, and *REMS* computes recall differently than any of the other tools. Notice that we contacted the authors and had extensive conversations with them to fully understand how to replicate their results best and how to use their tools so they perform the best.

Second, in order to bolster the robustness of our evaluation, we expand our evaluation to Extended Corpus, which replicates 1752 *EM* refactorings that took place in open-source projects. In this extended evaluation, we compare directly with JExtract. During our evaluation on Community Corpus, we noticed that GEMS and JExtract had the highest recall rates following *EM-Assist*, and surpassing all other tools. GEMS is an Eclipse plugin and is currently not functioning with the available versions of Eclipse. We contacted its authors for assistance, and indeed they confirmed that the tool has not been maintained, thus hindering the reproducibility of their results. Given that GEMS and JExtract's recall rates were remarkably similar and considering the extensive scale of the experiments, we opted for JExtract to execute and benchmark the recall rates against *EM-Assist*, using a Recall@5 metric at 3% tolerance.

2) ***Results****:* Table II shows the comparative effectiveness of *EM-Assist* in relation to the other six tools. We tame the non-determinism of LLMs as detailed in Section IV-B. Among the tools evaluated, GEMS ranked second highest, with JExtract closely behind. *EM-Assist* achieves superior performance with higher Recall@5 scores at 1% and 3% tolerance levels, recording values of 57.6% and 63%, respectively.

To further strengthen the validity of our results, we applied both *EM-Assist* and JExtract on the Extended Corpus that includes 1752 actual refactorings from open-source projects. For both tools, we calculated the recall for top-5 suggestions with a 3% tolerance value. As shown in Section IV-E2, *JExtract* achieved recall of 39.4%, **while *EM-Assist*'s mean recall was 53.4%, which is 35% improvement over the**

TABLE II: Evaluation results of *EM-Assist* with respect to six other tools on Community Corpus (upper table). The lower table compares *EM-Assist* against the best-in-class static analysis tool on the Extended Corpus.

	Recall@5		
Tool	Tolerance 1%	Tolerance 2%	Tolerance 3%
REMS	1.6%	1.6%	1.6%
GEMS	54.2%	**59.8%**	62.6%
JDeodorant	14.8%	18.4%	23.8%
JExtract	52.2%	59.3%	61.9%
SEMI	38.0%	47.0%	55.5%
LiveREF	10.6%	10.6%	13.1%
EM-Assist	**57.6%** ± 1.6	58.4% ± 1.6	**63%** ± 1.7

	Recall@5		
Tool	Tolerance 1%	Tolerance 2%	Tolerance 3%
JExtract	38.8%	39.3%	39.4%
EM-Assist	52.2% ± 0.9	52.7% ± 1	53.4% ±1

baseline. This shows that although *EM-Assist*'s advantage is modest when evaluated on a synthetic Community Corpus, its performance notably surpasses the baseline when it comes to replicating refactorings actually performed by developers.

To address the non-determinism of LLM outputs, we statistically analyzed the recall values of *EM-Assist* and JExtract over the Extended Corpus. We created a distribution of recall values for *EM-Assist* and then employed the Wilcoxon Signed-Rank Test to compare this distribution against the recall values of JExtract. The test rejected the null hypothesis, indicating a statistically significant difference in recall. Following this, we used the Hodges-Lehman estimator on each combination of distributions to quantify the difference in recall rates. The resultant value was 12.4 percentage points, suggesting a notable difference in performance between the two tools. This shows that *EM-Assist*'s improvements over the baseline are statistically significant.

When using an oracle of synthetic refactorings, *EM-Assist* has a slightly superior recall rate compared to its peers. When using an oracle of real-world refactorings, *EM-Assist*'s recall rate improvements over its peers are 35%, showing it better aligns with developer preferences.

F. RQ4: Usefulness of EM-Assist

EM-Assist is an interactive tool that provides maximum automation while still taking into account human input (see [33]). It presents refactoring suggestions to a developer, and based on their selection, *EM-Assist* applies the chosen refactorings and changes the code. Therefore, it is important to study the usefulness of these suggestions to developers and to gain insight into their decision-making process. We will take into account this feedback when releasing future versions.

1) ***Data and Experimental setup:*** To study the usefulness of the refactoring suggestions, we use the Firehouse survey research method [31]. A phenomenon and its solution are studied right after it happens (*e.g.*, observing victims' behavior right after a house catches fire). Accordingly, we engage with open-source developers immediately after they commit a long method into their repository, and we present them with suggestions for *EM* generated by *EM-Assist*. This direct interaction with developers, leveraging their recent familiarity with the code, ensures they provide reliable answers and are best equipped to evaluate the quality of suggestions. Notably, this method, as demonstrated by Silva et al. [14], boasts a significantly higher response rate compared to other survey-based studies that require developer participation.

To perform firehouse surveys, we engaged with developers contributing to the open-source projects *JetBrains/IntelliJ-Community* Edition [54] and *JetBrains/Runtime* [55]. The IntelliJ Community Edition, written in Java and Kotlin, has 15.8k stars, 417k commits, and 952 contributors. JetBrains/Runtime, written in Java, has 969 stars, 76k commits, and 806 contributors. These projects exemplify significant engagement, maturity, and attention to code quality, making them ideal for our study.

Firehouse Survey Method: We monitored the projects daily, analyzing every new commit to identify newly added long methods. To enhance this, we extended RefactoringMiner [6] to prevent mistakenly identifying refactored methods (*e.g.*, renamed or moved) as newly added. Once we identified a long method, we contacted the developer. To minimize the intrusion for the developers, we only asked their opinion for one single long method (regardless of how many methods they authored) and only presented three suggestions for their method (regardless of how many *EM-Assist* generated).

Then we asked developers for their level of agreement with each suggestion on a 6-point Likert scale ranging from Strongly Agree to Strongly Disagree. Additionally, we encouraged them to share their reasoning, detailing what aspects they liked or disliked about the suggestions or any changes they would consider. As developers answered with free-form text, we conducted thematic analysis to interpret these responses, adhering to established best practices [56]–[58].

2) ***Results:*** We surveyed 20 developers, out of which 16 responded, bringing us to a 80% response rate. This response rate is notably higher than the typical response rate in questionnaire-based software engineering surveys, which typically hovers around 5% [59], and this can be attributed to using firehouse surveys. Table III shows the agreement levels. For each developer survey, we present the entry for the highest-rated suggestion by the developer. Table III shows that even when giving just three ranked suggestions per method so that we do not overwhelm the developer, 81.3% of respondents find them useful (i.e., chose a positive rating). Thus, *EM-Assist* generates useful refactorings that align with developer preferences. As *EM-Assist* finds improvement opportunities and generates useful suggestions even for high-quality projects like the ones we used, we are confident it would discover useful refactoring opportunities in regular-quality projects.

Developers remarked that these refactoring suggestions significantly enhance code quality. They appreciated a fresh perspective on their code: "*... these suggestions made me look at this code with new eyes once more, and I will try to refactor it*". Furthermore, developers strongly desire to see *EM-Assist* in production: "*Thank you for interesting suggestions! Hope*

TABLE III: Developers' levels of agreement to the refactoring suggestions produced by *EM-Assist*

Strongly Disagree	Disagree	Somewhat Disagree	Somewhat Agree	Agree	Strongly Agree
2	1	0	5	6	2

to see this in production in the future." These encouraging comments highlight the positive impact and usefulness of *EM-Assist*'s recommendations in the daily development process.

The top-3 reasons why developers did not accept suggestions:

1) Splitting a monolithic algorithm into smaller parts could potentially complicate code organization, and the original method is concise enough. It suggests that in the future, *EM-Assist* should more accurately take the developer's method size preferences into consideration.

2) The extracted method has too many parameters. This shows the importance of implementing additional filters based on the number of parameters in the suggestion.

3) The extracted method does not promote reuse: While this is a valid concern, it is important to note that *EM-Assist* focuses solely on individual methods as input and does not take into account broader contextual factors, such as a file or project-level considerations. In the future, we could extend *EM-Assist*'s scope to suggest extract methods for reuse at higher levels, such as file, module, or project levels [60], [61].

Developers say 81% of *EM-Assist*'s suggestions are useful.

V. Threats To Validity

1) **Internal Validity**: *Does our work produce valid results? EM-Assist*'s effectiveness can vary depending on the oracle used for evaluation. To mitigate this, we employ two datasets for assessment: a publicly available, independent dataset commonly utilized by other researchers, and a more extensive dataset derived from open-source repositories. Our thorough evaluation of *EM-Assist* on these datasets reveals its enhanced recall capabilities, outperforming other state-of-the-art tools. Furthermore, the firehouse survey results also show that open-source developers found *EM-Assist*'s suggestions useful.

2) **External Validity**: *Do our results generalize? EM-Assist*'s effectiveness hinges on the chosen LLM model. As LLMs continue to advance, especially with upcoming iterations trained on more extensive datasets, we anticipate further improvements in outcomes. Although we have thoroughly evaluated our tool with only GPT-3.5-turbo, and expect different LLMs to behave slightly differently, the architecture of *EM-Assist* allows for effortless adaptation to future LLMs. Lastly, it is important to highlight that while our approach is conceptually language-agnostic, our current *EM-Assist* implementation is tailored for Java and Kotlin. Therefore, we cannot extrapolate our performance results to programs written in other languages.

LLM responses are non-deterministic, especially as their temperature hyper-parameter is increased, which can pose challenges for the reproducibility of results. To handle this, we use two techniques while evaluating *EM-Assist*. First, we re-prompt the LLM multiple times to account for a significant amount of its possible responses. The rest of *EM-Assist*'s pipeline works with the superset of responses to find valid *EM* suggestions. Second, we back our results with statistical analysis. After performing multiple prompts, we resample our data multiple times to compute mean and standard deviation for recall. This increases the confidence in our results, as we are able to use statistical tests to back up our claims.

3) **Verifiability**: We make *EM-Assist*, its source code, the exact LLM prompt we use and examples of LLM responses, the datasets and results, publicly available [32].

VI. Related Work

We observed two primary approaches for *EM* refactoring: (i) tools designed to recommend specific code fragments for developers to refactor using *EM*, and (ii) tools that predict whether a host method needs *EM* refactoring. Accordingly, the related work is structured into two sections.

Suggesting EM refactoring: Several studies [4]–[6], [62] indicate that EM refactoring is among the top five most frequently performed practices, leading to the development of numerous supporting tools. JDeodorant [1] automatically calculates block-based program slicing for variables in assignment statements. Given a slicing criterion as input, JDeodorant can extract relevant statements while preserving program behavior. JExtract [28] operates based on the block structure, where each block structure contains a group of statements organized with a linear control flow. Given a method, JExtract detects involved block structures and heuristically ranks them as refactoring candidates. SEMI [8] explores the coherence between statements and returns code fragments with high cohesion as refactoring candidates. LiveRef [30] is an IntelliJ plugin that offers real-time *EM* refactoring suggestions and immediate application, guided by code quality metrics and visual cues in the code editor. It continually adapts to code changes, providing live refactoring support. REMS [12] utilizes multi-view representations from the code property graph. It then trains a ML classifier to guide the extraction of suitable lines of code as a new method. GEMS [11] encodes metrics related to complexity, cohesion, and coupling as features to train ML classifiers for recommending EM refactoring opportunities. Other tools propose EM refactoring based on program slicing [7], [43], [63], separation of concerns, and the single responsibility principle [8], while some [60], [64]–[66] aim to reduce code clones and duplication.

In contrast, our work pioneers the integration of LLMs with static analysis techniques to scrutinize and refine the outputs of LLMs, compensating for their lack of understanding in program semantics, thereby facilitating the correct and safe execution of EM refactoring.

Predicting EM Refactoring: Researchers identified the necessity for implementing EM refactoring. Aniche et al. [67] evaluated the effectiveness of various ML algorithms for predicting software refactorings, including EM refactoring. Their models were trained on thousands of refactorings mined with

RefactoringMiner [6] from open-source projects. Van der Leij et al. [68] replicated the study [67] at the ING company and discovered that ML models predict very well the opportunity for applying EM refactoring. Others [69], [70] use commit messages to predict software refactorings. While these tools are very good at predicting the type of refactoring a method needs to undergo, they complement nicely with *EM-Assist* and its ability to suggest code fragments to be extracted and also to apply the refactoring.

VII. Conclusions

EM-Assist is the first refactoring tool that exploits the untapped potential of LLMs for refactoring tasks. AI systems can break down at unexpected places. For an LLM, this results in refactoring suggestions that seem plausible at first reading but are actually deeply flawed. Our experiments show that LLMs are not reliable and need to be checked. We discovered a novel way of checking LLM results and making them useful for refactoring tasks.

EM-Assist recommends *EM* refactorings that align with developers' preferences. This is evidenced when replicating thousands of real-world refactoring scenarios from open-source repositories. Moreover, our firehouse surveys with developers of high-quality codebases that authored recent changes revealed that 81.3% of respondents found *EM-Assist*'s suggestions useful.

We discovered a new set of best practices when using LLMs for refactoring. One of the biggest challenges with LLMs is taming non-determinism. Unlike in traditional static analysis tools where we avoid redundant computations, working effectively with LLMs requires re-prompting (asking the same question several times) and creating a superset of all suggestions. This was key to get the most out of the LLM. Thus, we designed novel ranking methods to take into account the LLM workflow. Moreover, we learned that the more precise the prompt, the higher the quality of the suggestions. Few-shot learning worked best for refactoring tasks.

EM-Assist provides maximum automation of the full lifecycle of employing an LLM (i.e., prompting, validating, and enhancing suggestions) and it executes refactorings correctly within the IDE while still keeping the programmer in the loop as the ultimate decision maker. This ushers a new era when AI assistants do not take over the programmers, but become effective companions for code renovation tasks. Together, programmers, assisted by AI and the IDE, go further.

With the emergence of LLMs, researchers push the boundaries of current automation tools. One notable example is the possibility to tackle another longstanding challenge in *EM* refactoring: extracting related but non-contiguous lines into a single method. This has been a challenge for decades, and now LLMs (together with static analysis tools) can offer a viable solution. We hope others continue to further expand the automation support for *EM*.

In this work, we focused on *EM* refactoring for Java and Kotlin, but our approach can be expanded to other refactoring automation tasks, and to other programming languages. Indeed, our emerging results following the work presented in this publication show that our approach is very effective in handling dozens of other refactoring kinds beyond *EM*. Moreover, we are currently extending support to include dynamically-typed languages such as Python. We hope that the best practices that we discovered for using LLMs effectively for refactoring tasks inspire others to further advance the field of refactoring.

VIII. Open Science Policy

We have made *EM-Assist* and evaluation data available on our website [32], and the tool is freely available to be installed from the JetBrains Marketplace [33].

IX. Acknowledgements

We thank the ML Methods in Software Engineering Lab at JetBrains Research, and the ICSME-2024 reviewers for their insightful and constructive feedback for improving the paper. This research was partially funded through the NSF grants CNS-1941898, CNS-2213763, and the Industry-University Cooperative Research Center on Pervasive Personalized Intelligence.

References

[1] N. Tsantalis and A. Chatzigeorgiou, "Identification of extract method refactoring opportunities for the decomposition of methods," *Journal of Systems and Software*, 2011. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121211001191

[2] R. D. Banker, S. M. Datar, C. F. Kemerer, and D. Zweig, "Software complexity and maintenance costs," *Communications of the ACM*, 1993.

[3] M. Fowler, *Refactoring: Improving the Design of Existing Code*, 1999.

[4] S. Negara, N. Chen, M. Vakilian, R. E. Johnson, and D. Dig, "A comparative study of manual and automated refactorings," in *ECOOP 2013 – Object-Oriented Programming*, G. Castagna, Ed., 2013.

[5] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *IEEE Transactions on Software Engineering*, 2012.

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

[7] K. Maruyama, "Automated method-extraction refactoring by using block-based slicing," in *Proceedings of the 2001 Symposium on Software Reusability: Putting Software Reuse in Context*, ser. SSR '01, 2001. [Online]. Available: https://doi.org/10.1145/375212.375233

[8] S. Charalampidou, A. Ampatzoglou, A. Chatzigeorgiou, A. Gkortzis, and P. Avgeriou, "Identifying extract method refactoring opportunities based on functional relevance," *IEEE Transactions on Software Engineering*, 2017.

[9] L. Yang, H. Liu, and Z. Niu, "Identifying fragments to be extracted from long methods," in *Proceedings of the 2009 16th Asia-Pacific Software Engineering Conference*, ser. APSEC '09, 2009. [Online]. Available: https://doi.org/10.1109/APSEC.2009.20

[10] O. Tiwari and R. Joshi, "Identifying extract method refactorings," in *15th Innovations in Software Engineering Conference*, ser. ISEC 2022, 2022. [Online]. Available: https://doi.org/10.1145/3511430.3511435

[11] S. Xu, A. Sivaraman, S.-C. Khoo, and J. Xu, "Gems: An extract method refactoring recommender," in *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, 2017.

[12] D. Cui, Q. Wang, S. Wang, J. Chi, J. Li, L. Wang, and Q. Li, "Rems: Recommending extract method refactoring opportunities via multi-view representation of code property graph," in *2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC)*, 2023.

[13] R. C. Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*, 1st ed., 2017.

[14] D. Silva, N. Tsantalis, and M. T. Valente, "Why we refactor? confessions of github contributors," ser. FSE 2016, 2016. [Online]. Available: https://doi.org/10.1145/2950290.2950305

[15] S. Fakhoury, D. Roy, A. Hassan, and V. Arnaoudova, "Improving source code readability: Theory and practice," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 2019.

[16] S. Scalabrino, G. Bavota, C. Vendome, M. Linares-Vasquez, D. Poshyvanyk, and R. Oliveto, "Automatically assessing code understandability," *IEEE Transactions on Software Engineering*, 2019.

[17] M. Dilhara, A. Ketkar, and D. Dig, "Understanding software-2.0: A study of machine learning library usage and evolution," *ACM Trans. Softw. Eng. Methodol.*, jul 2021. [Online]. Available: https://doi.org/10.1145/3453478

[18] M. Dilhara, A. Ketkar, N. Sannidhi, and D. Dig, "Discovering repetitive code changes in Python ML systems," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22, 2022. [Online]. Available: https://doi.org/10.1145/3510003.3510225

[19] M. Dilhara, D. Dig, and A. Ketkar, "Pyevolve: Automating frequent code changes in python ml systems," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 995–1007. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00091

[20] M. Dilhara, "Discovering repetitive code changes in ml systems," ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 1683–1685. [Online]. Available: https://doi.org/10.1145/3468264.3473493

[21] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[22] OpenAI, "Gpt-4 technical report," 2023. [Online]. Available: https://arxiv.org/pdf/2303.08774.pdf

[23] GoogleAI, "Palm 2," 2024. [Online]. Available: https://ai.google/discover/palm2/

[24] M. Ciniselli, N. Cooper, L. Pascarella, A. Mastropaolo, E. Aghajani, D. Poshyvanyk, M. Di Penta, and G. Bavota, "An empirical study on the usage of transformer models for code completion," *IEEE Transactions on Software Engineering*, 2022.

[25] E. Aleksandra, S. Yaroslav, B. Egor, G. Yaroslav, D. Danny, and B. Timofey, "From commit message generation to history-aware commit message completion," ASE 2023. [Online]. Available: https://arxiv.org/pdf/2308.07655.pdf

[26] F. Sidong and C. Chunyang, "Prompting is all your need: Automated android bug replay with large language models," in *Proceedings of the 46th International Conference on Software Engineering*, ser. ICSE '24, 2024.

[27] M. Dilhara, A. Bellur, D. Dig, and T. Bryksin, "Unprecedented Code Change Automation: The Fusion of LLMs and Transformation by Example," in *32nd ACM Symposium on the Foundations of Software Engineering (FSE '24)*, ser. FSE 2024, 2024, to appear. [Online]. Available: https://doi.org/10.1145/3643755

[28] D. Silva, R. Terra, and M. T. Valente, "Jextract: An eclipse plug-in for recommending automated extract method refactorings," *arXiv preprint arXiv:1506.06086*, 2015.

[29] S. Fernandes, A. Aguiar, and A. Restivo, "A live environment to improve the refactoring experience," in *Companion Proceedings of the 6th International Conference on the Art, Science, and Engineering of Programming*, 2022.

[30] S. Fernandes, A. Aguiar, and Restivo, "LiveRef: A Tool for Live Refactoring Java Code," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22, 2023. [Online]. Available: https://doi.org/10.1145/3551349.3559532

[31] E. Murphy-Hill, T. Zimmermann, C. Bird, and N. Nagappan, "The design space of bug fixes and how developers navigate it," *IEEE Transactions on Software Engineering*, 2015.

[32] JetBrains, "EM-Assist source code and datasets," 2024. [Online]. Available: https://github.com/llm-refactoring/llm-refactoring-plugin

[33] ——, "EM-Assist installation from JetBrains Marketplace," 2024. [Online]. Available: https://plugins.jetbrains.com/plugin/23403-llm-extract-function

[34] ——, "EM-Assist demo video and screencast," 2024. [Online]. Available: https://youtu.be/3E6KsHAg3js

[35] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., 2020. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

[36] J. Wei, X. Wang, D. Schuurmans, M. Bosma, b. ichter, F. Xia, E. Chi, Q. V. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," in *Advances in Neural Information Processing Systems*, 2022. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2022/file/9d5609613524ecf4f15af0f7b31abca4-Paper-Conference.pdf

[37] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, 2019.

[38] S. Gao, X.-C. Wen, C. Gao, W. Wang, H. Zhang, and M. R. Lyu, "What makes good in-context demonstrations for code intelligence tasks with llms?" in *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '38. ACM, ASE 2023. [Online]. Available: https://arxiv.org/abs/2304.07575

[39] Anonymous. (2024) Prompt used in the experiment. [Online]. Available: https://github.com/llm-refactoring/llm-refactoring-plugin/tree/main#prompt-example

[40] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2020.

[41] R. Haas and B. Hummel, "Deriving extract method refactoring suggestions for long methods," in *International Conference on Software Quality*. Springer, 2015.

[42] L. Yang, H. Liu, and Z. Niu, "Identifying fragments to be extracted from long methods," in *2009 16th Asia-Pacific Software Engineering Conference*. IEEE, 2009.

[43] A. Abadi, R. Ettinger, and Y. Feldman, "Fine slicing for advanced method extraction," in *3rd workshop on refactoring tools*, 2009.

[44] P. Meananeatra, S. Rongviriyapanish, and T. Apiwattanapong, "Refactoring opportunity identification methodology for removing long method smells and improving code analyzability," *IEICE TRANSACTIONS on Information and Systems*, 2018.

[45] B. E. Cossette and R. J. Walker, "Seeking the ground truth: A retroactive study on the evolution and migration of software libraries," ser. FSE '12, 2012. [Online]. Available: https://doi.org/10.1145/2393596.2393661

[46] T. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, 1976.

[47] M. H. Halstead, *Elements of Software Science (Operating and programming systems series)*. USA: Elsevier Science Inc., 1977.

[48] Wikipedia. (2024) Bootstrapping. [Online]. Available: https://en.wikipedia.org/wiki/Bootstrapping_(statistics)

[49] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Software Testing, Verification and Reliability*, vol. 24, no. 3, pp. 219–250, 2014.

[50] T. L. Scao, A. Fan, C. Akiki, E. Pavlick, S. Ilić, D. Hesslow, R. Castagné, A. S. Luccioni, F. Yvon, M. Gallé *et al.*, "Bloom: A 176b-parameter open-access multilingual language model," *arXiv preprint arXiv:2211.05100*, 2022.

[51] Anthropic, "Introducing claude," 2023. [Online]. Available: https://www.anthropic.com/index/introducing-claude

[52] Meta, "Introducing llama," 2023. [Online]. Available: https://ai.meta.com/llama/

[53] Falcon, "Falcon," 2023. [Online]. Available: https://falconllm.tii.ae

[54] JetBrains, "Intellij community edition," 2023. [Online]. Available: https://github.com/JetBrains/intellij-community

[55] J. Runtime, "Jetbrains runtime," 2024. [Online]. Available: https://github.com/JetBrains/JetBrainsRuntime

[56] D. S. Cruzes and T. Dybå, "Research synthesis in software engineering: A tertiary study," *Information and Software Technology*, 2011, special Section on Best Papers from XP2010. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S095058491100005X

[57] V. Braun and V. Clarke, "Using thematic analysis in psychology," *Qualitative Research in Psychology*, 2006. [Online]. Available: https://www.tandfonline.com/doi/abs/10.1191/1478088706qp063oa

[58] J. L. Campbell, C. Quincy, J. Osserman, and O. K. Pedersen, "Coding in-depth semistructured interviews: Problems of unitization and intercoder reliability and agreement," *Sociological Methods & Research*, 2013.

[59] J. Singer, S. E. Sim, and T. C. Lethbridge, *Software Engineering Data Collection for Field Studies*, 2008. [Online]. Available: https://doi.org/10.1007/978-1-84800-044-5_1

[60] E. Alomar, A. Ivanov, Z. Kurbatova, Y. Golubev, M. W. Mkaouer, A. Ouni, T. Bryksin, L. Nguyen, A. Kini, and A. Thakur, "Just-in-time code duplicates extraction," *Information and Software Technology*, 02 2023.

[61] Alomar, A. Ivanov, Z. Kurbatova, Y. Golubev, M. W. Mkaouer, A. Ouni, T. Bryksin, L. Nguyen, A. Kini, and A. Thakur, "Anticopypaster: Extracting code duplicates as soon as they are introduced in the ide," 01 2023.

[62] E. Murphy-Hill and A. P. Black, "Breaking the barriers to successful refactoring: observations and tools for extract method," in *Proceedings of the 30th international conference on Software engineering*, 2008.

[63] A. Lakhotia and J.-C. Deprez, "Restructuring programs by tucking statements into functions," *Information and Software Technology*, 1998.

[64] R. Tairas and J. Gray, "Increasing clone maintenance support by unifying clone detection and refactoring activities," *Information and Software Technology*, 2012.

[65] R. Yue, Z. Gao, N. Meng, Y. Xiong, X. Wang, and J. D. Morgenthaler, "Automatic clone recommendation for refactoring based on the present and the past," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018.

[66] N. Yoshida, S. Numata, E. Choiz, and K. Inoue, "Proactive clone recommendation system for extract method refactoring," in *2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR)*. IEEE, 2019.

[67] M. Aniche, E. Maziero, R. Durelli, and V. H. Durelli, "The effectiveness of supervised machine learning algorithms in predicting software refactoring," *IEEE Transactions on Software Engineering*, 2020.

[68] D. van der Leij, J. Binda, R. van Dalen, P. Vallen, Y. Luo, and M. Aniche, "Data-driven extract method recommendations: a study at ing," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021.

[69] E. A. AlOmar, J. Liu, K. Addo, M. W. Mkaouer, C. Newman, A. Ouni, and Z. Yu, "On the documentation of refactoring types," *Automated Software Engineering*, 2022.

[70] P. S. Sagar, E. A. AlOmar, M. W. Mkaouer, A. Ouni, and C. D. Newman, "Comparing commit messages and source code metrics for the prediction refactoring activities," *Algorithms*, 2021.